Breaking and Fixing Private Set Intersection Protocols

Mikkel Lambæk, 20113601

Concat at: mikkel.lambaek@post.au.dk

Master's Thesis, Computer Science

June 2016

Advisers: Claudio Orlandi, Ivan Damgård

AARHUS
UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Abstract

A private set intersection protocol consists of two parties, a Sender and a Receiver, each with a secret input set. The protocol aims to have the Receiver output an intersection of the two sets while keeping the elements in the sets secret.

This thesis thoroughly analyzes four recently published set intersection protocols, where it explains each protocol and checks whether it satisfies its corresponding security definition.

The first two protocols [PSZ14, PSSZ15a] use random oblivious transfer where [PSSZ15a] is an optimized version of [PSZ14]. In the optimized protocol a correctness error is identified and prevented at a minor increase in run-time. An attempt to make [PSSZ15a] secure against a malicious adversary is shown, where the resulting protocol is proven secure against a semi-honest Sender and malicious Receiver.

The third protocol [DCW13] is based on Bloom filters combined with oblivious transfer, and proposes protocols for two different security levels. The semi-honestly secure protocol satisfies its definition, while their proposal for a maliciously secure protocol is insufficient. This thesis shows two attacks a malicious Sender is capable of, without finding efficient countermeasures.

The last protocol [DC16] allows computing four different set operations, where five errors are identified. Each error is explained and a proposal to avoid the issue is shown.

Acknowledgements

First of all I would like to especially thank Claudio Orlandi for helpful guidance and many interesting discussions. Working with him has been a great experience.

Secondly I would like to Ivan Damgård for standing in for Claudio during the defence of this thesis, and at last I would like to thank Stefan Lambæk for reading through my thesis, and providing feedback.

Mikkel Lambæk,
Aarhus, June 29, 2016.

Contents

Chapter 1

Introduction

Cryptographers have studied multi-party computation for several years by trying to create general protocols that can compute any function. The performance of these general protocols has been lacking, resulting in the study of special purpose protocols to compute a specific function, to obtain better run-times.

One such function is private set intersection (PSI) which has been studied by various authors. A private set intersection protocol consists of two parties, a Sender and a Receiver, each having a set as input, which respectively are denoted X and Y. Together they want to compute the intersection of their sets, $X \cap Y$, without revealing elements not contained in the intersection. Usually the Receiver will learn $X \cap Y$ and $|X|$ without learning $X \setminus Y$, while the Sender learns $|Y|$ and nothing else.

There are many real-world applications for PSI, where one example of that is two national law enforcement bodies each having a list of suspected terrorists. Due to national laws they may not be allowed to disclose their full lists, even when collaborating. Using a PSI protocol the two agencies can find targets of common interest, which their laws might allow.

Equivalently one of the agencies may want to check foreign planes for potential terrorists, again without revealing its confidential list. A PSI protocol can be used on the terrorist list and passenger list, and if an empty intersection is returned the plane should be allowed to land. If a non-empty intersection is returned alternative action may be required.

During the last decade a significant decrease in the run-time of PSI protocols has been achieved, to the point where several linear-time protocols have been developed against both semi-honest and malicious adversaries. In this thesis an in-depth look is taken at four recently proposed PSI protocols. It shows various issues in three of the papers, and if a proof is not included in the paper one is created as part of this thesis.

Chapter 2: The preliminaries presents various concepts necessary for the following chapters. It starts by introducing the general notation used through this thesis, followed by defining secret sharing schemes and two instantiations of

it. Afterwards two types of encryption schemes are defined, namely symmetric-key encryption and the requirements for it to be secure, followed by defining homomorphic encryption, and its necessary properties.

The four PSI protocols use multiple different data structures, where all of them will be explained here. In particular four different Cuckoo hashing algorithms and multiple variants of Bloom filters are being used. Following this the two standard definitions of security are provided, and oblivious transfer is explained. At last a section presenting what private set intersection is, and in particular what leakage it allows and the PSI specific notation used in the remaining chapters.

Chapter 3: In this chapter an in-depth look will be taken at the semi-honestly secure private set intersection protocol from [PSZ14]. Due to the protocol's complexity this chapter thoroughly explains one way to construct PSI based on oblivious transfer, where two different hashing schemes are used to increase efficiency. This chapter is partially in preparation for Chapter 4, which introduces three optimizations to [PSZ14]. At the end a proof of security is also created, as one was not included in [PSZ14].

Chapter 4: This chapter consists of two parts. The first is understanding the optimizations provided by [PSSZ15a] to obtain a more efficient semi-honestly secure PSI protocol. This optimized version is analyzed which results in an easily fixed correctness error, and the slightly modified protocol is then proven correct, as the proof was omitted from [PSSZ15a].

Secondly the slightly modified protocol is analyzed with a malicious adversary, which results in identifying three vulnerabilities. Suggestions to prevent two of these issues are provided, which results in a protocol that is secure against a semi-honest Sender and a malicious Receiver. At the end this new protocol is proven secure against a malicious Receiver.

Chapter 5: In this chapter the two protocols from [DCW13] are analyzed, where both are based on the paper's new data structure, called garbled Bloom filters. The first protocol is secure against a semi-honest adversary, and is primarily given as preliminary understanding for the paper's second protocol - an enhanced version of the first protocol. The enhanced version is meant to be secure against a malicious adversary, but as will be shown a corrupt Sender can perform two different attacks. The first is a selective failure attack, which causes the Receiver to either abort and output $\perp$ or finish the protocol and output an intersection including a specific element. The second attack makes the Sender's input dependant on the Receiver's input, where no efficient countermeasures have been found to either attack.

Chapter 6: In this chapter the semi-honestly secure set operation protocols from [DC16] will be analyzed. The four protocols they suggest are based on partially homomorphic encryption combined with Bloom filters, where five issues will be presented, where potential countermeasures will be suggested for each issue.

The first issue relates to their security proof, which relies on a different property than they assert. The second issue relates to lacking randomness, allowing the Receiver to find the intersection in a set union protocol. The third issue is their use of multiplicative homomorphic encryption to obtain a PSI protocol. The fourth issue is leakage during their cardinality protocols, and the last issue is leaking an upper bound on the Receiver's input size as their protocol is meant to hide it.

The papers and their vulnerabilities can be seen in Table 1.1. The authors of the papers have been notified of most issues, and as such are working on fixing them. The exact versions analyzed during this thesis are included in references [PSZ14], [PSSZ15b] (which is the full version of [PSSZ15a]), [DCW13] and [DC16].

Paper	Adversary used	Analyzed with	Vulnerability	Fixed?
[PSSZ15a]	Semi-honest	Semi-honest	Correctness issue	Yes
[PSSZ15a]	Semi-honest	Malicious	Three issues	Two
[DCW13]	Malicious	Malicious	Selective failure	No
[DCW13]	Malicious	Malicious	Input dependence	No
[DC16]	Semi-honest	Semi-honest	Proof	Yes
[DC16]	Semi-honest	Semi-honest	Lacking randomness	Yes
[DC16]	Semi-honest	Semi-honest	PSI protocol	Yes
[DC16]	Semi-honest	Semi-honest	Cardinality protocols	Yes
[DC16]	Semi-honest	Semi-honest	Upper bound on $\|Y\|$	Yes

Table 1.1: The papers, their claimed security and the adversary with which it was analyzed. Includes the identified vulnerabilities and whether it was fixed.

Chapter 2

Preliminaries

2.1 Notation

Let the security parameter be κ, and the statistical security parameter be λ. Throughout this thesis it will be assumed that $\kappa = 128$-bit security, and $\lambda = 40$. A function $\mu(\cdot)$ is said to be negligible in κ, or just negligible, if for every positive polynomial $p(\cdot)$ and all sufficiently large κ's it holds that $\mu(\kappa) < 1/p(\kappa)$. Let PPT be shorthand for probabilistic polynomial time, and let $\in_{\mathrm{R}}$ mean a uniformly random choice in a specified set. Let $\perp$ denote empty output. A vector $\boldsymbol{v} = (v_1, \ldots, v_m)$ is said to be balanced if for every i, j it holds that $|v_i| = |v_j|$. The weight of a vector $\boldsymbol{v}$ is denoted $HammingWeight(\boldsymbol{v})$, and is the number of indices where $v_i \neq \perp$. Given an l-bit string s its indices are from 0 to $l-1$, and $s[m;n]$ means the bits from m to $n-1$, where $0 \leq m < n \leq l$.

A probability ensemble $X = \{X(a,\kappa)\}_{a\in\{0,1\}^*;\kappa\in\mathbb{N}}$ is an infinite sequence of random variables indexed by a and $\kappa \in \mathbb{N}$. Two probability ensembles X, Y are said to be computationally indistinguishable, denoted $X \overset{\mathrm{c}}{\equiv} Y$, if for every non-uniform PPT algorithm D there exists a negligible function $\mu(\cdot)$ such that for every $a \in \{0,1\}^*$ and every $\kappa \in \mathbb{N}$,

$$\big|Pr[D(X(a,\kappa)) = 1] - Pr[D(Y(a,\kappa)) = 1]\big| \leq \mu(\kappa)$$

2.1.1 Lagrange Interpolation

Lagrange interpolation is based on the fundamental theorem of algebra, which states that given any arbitrary t points on a polynomial of degree $t-1$ it is possible to reconstruct the original polynomial, and given $t-1$ points on the same polynomial nothing is known about it. A definition of Lagrange interpolation is stated in Theorem 2.1, and will be denoted $f \leftarrow LagrangeInterpolation(\boldsymbol{v})$, where $\boldsymbol{v}$ is a vector of length $n \geq t$ where each coordinate is either a point or $\perp$, to signify no knowledge of this coordinate.

THEOREM 2.1 (LAGRANGE INTERPOLATION, [DEM12]) *Let* $\{x_j, j = 0, \ldots, N\}$ *be a collection of disjoint numbers in a field* $\mathbb{F}$. *Let* $\{y_j, j = 0, \ldots, N\}$ *be a collection of numbers in* $\mathbb{F}$. *Then there exists a polynomial* $f_N \in \mathbb{F}_N$ *such that*

$$f_N(x_j) = y_j, j = 0, \ldots, N$$

Its expression is

$$f_N(x) = \sum_{k=0}^{N} y_k \cdot L_k(x)$$

where $L_k(x)$ *are the Lagrange elementary polynomials*

2.1.2 Secret Sharing Scheme

A (t, n)-threshold secret sharing scheme is an algorithm that, given secret input s, is able to split s into n shares that can be distributed. The output need to be constructed such that t of these shares can uniquely reconstruct s, while $t-1$ shares provide no knowledge of s. This has been formally stated in Definition 2.2.

DEFINITION 2.2 (SECRET SHARING SCHEME) *A* (t, n)*-threshold secret sharing scheme encodes the secret* s *into* n *shares* $(s_1, \ldots, s_n)$ *such that* $< t$ *shares yield no information about* s, *and given* $\geq t$ *shares* s *can be efficiently and uniquely computed.*

Shamir's Secret Sharing Scheme

[Sha79] proposes a (t, n)-threshold secret sharing scheme based on polynomials, using Lagrange interpolation. The pseudocode for sharing and reconstruction can be seen in Figure 2.1. Intuitively it assumes, without loss of generality, that a secret can be represented as an integer denoted s. By finding a uniformly random polynomial p of degree $t-1$ under the constraint that $p(0) = s$ it is possible to share the n points $(p(1), \ldots, p(n))$, where at least t are required to learn s. This correctness and secrecy is trivially implied by Lagrange interpolation.

To ensure correctness and security Shamir's secret sharing scheme has two requirements to the used field. First of all the field size needs to be a prime number q, to guarantee proper randomness within the field. Secondly it needs the prime number q to be larger than n, as $p(n)$ would no longer be a unique number - two shares $p(n)$ and $p(n \mod q)$ would otherwise be equivalent, which means the reconstruction cannot be done with any $\geq t$ shares.

1: **procedure** SSSS.$\textsc{Share}_{t,n}$(s)
2: Choose polynomial $p_s(X) \in_{\mathrm{R}} \mathbb{F}_t[X]$, where $deg(p) = t - 1$ and $p(0) = s$
3: **return** $\boldsymbol{v} = (p(1), \ldots, p(n))$

1: **procedure** SSSS.$\textsc{Reconstruct}_{t,n}(\boldsymbol{v})$
2: **if** $HammingWeight(\boldsymbol{v}) < t$ **then return** $\perp$
3: $p \leftarrow LagrangeInterpolation(\boldsymbol{v})$
4: **return** $p(0)$

Figure 2.1: Pseudocode for Shamir's Secret Sharing Scheme

XOR-based Secret Sharing Scheme

A simple (n, n)-threshold secret sharing scheme is based upon the XOR operator. A secret s of length k can be shared by generating $n-1$ uniformly random shares $(v_1, \ldots, v_{n-1})$, where each v_i is of length k, followed by calculating the last share as $v_n = s \oplus v_1 \oplus \cdots \oplus v_{n-1}$.

Reconstructing s clearly requires knowing all n shares, as even given $n-1$ shares nothing is learned of the last share, since the last share is still uniformly random in $\{0,1\}^k$.

The pseudocode for sharing and reconstructing can be seen in Figure 2.2.

1: **procedure** XOR.$\textsc{Share}_{n,n}$(s)
2: Choose $(v_1, \ldots, v_{n-1})$ uniformly at random
3: Calculate $v_n = s \oplus v_1 \oplus \cdots \oplus v_{n-1}$
4: **return** $\boldsymbol{v} = (v_1, \ldots, v_n)$

1: **procedure** XOR.$\textsc{Reconstruct}_{n,n}(\boldsymbol{v})$
2: **if** $HammingWeight(\boldsymbol{v}) < n$ **then return** $\perp$
3: $s = v_1 \oplus \cdots \oplus v_n$
4: **return** s

Figure 2.2: Pseudocode for XOR-based Secret Sharing Scheme

2.1.3 Random Oracle Model

The Random Oracle Model (ROM) is formally introduced by [BR93] as an efficient method to model hash functions that output uniformly random values while being consistent, meaning invoking it twice on the same input will produce the same output twice. This is done by viewing the oracle as a black box with a list of pairs (s_i, r_i), where all s_i's are the inputs it has previously been invoked on and the r_i's are the corresponding k-bit outputs. When the oracle is invoked on input s it checks whether s is contained in its list, in which case it outputs the relevant r_i, and otherwise it generates a new uniformly random k-bit value r, saves the pair (s, r) in its list and outputs r. This is denoted $H : \{0,1\}^j \to \{0,1\}^k$ for some j and k.

When a random oracle is used in a proof there exists three ways to model it, the standard version, one with extractability and one with programmability. In the standard version the adversary knows nothing about the honest party's

queries, as the oracle is used as a third independent party. The two other properties are related to proofs by reduction, which means a reduction obtains the queries of the adversary, and either has to forward the query to an oracle or answer the queries himself. Extractability is being used when the reduction forwards the query, while obtaining information of the adversary's queries (and the results). Programmability is referred to when the reduction answers the query, under the constraint that the output looks uniformly random.

Extractability and programmability cannot be modelled by any known real-world hash function, which means those properties provide a weaker sense of security. All random oracles used in this thesis are in the standard model.

2.2 Encryption schemes

During this thesis two different types of encryption schemes will be used, both of which will be defined here. First symmetric key, followed by partially homomorphic encryption scheme.

2.2.1 Symmetric Key Encryption Scheme

Symmetric key, or secret key, encryption schemes are used as a secure connection for communication between two parties that have a private piece of information called the secret key, k. Using this pre-shared key two functions E and D are defined to map a message m to a ciphertext c and back again, without failure. Definition 2.3 formally describes this concept, while also defining a generator algorithm for the key.

DEFINITION 2.3 (SECRET-KEY ENCRYPTION SCHEME, [GOL04]) *A* secret-key encryption scheme *is a triple (G, E, D), of PPT algorithms with the following properties:*

- *On input 1^κ, algorithm G (called the* key-generator*) outputs a bit string.*
- *For every string k in the range of $G(1^\kappa)$, and for every $m \in \{0,1\}^*$, algorithms E (encryption) and D (decryption) satisfy*

$$Pr[D(k, E(k, m)) = m] = 1$$

where the probability is taken over the internal coin tosses of algorithms E and D

Definition 2.3 says nothing about obfuscating the message m,[1] which is why a second property is required to model indistinguishability of plaintexts.

Obfuscation is obtained through IND-CPA, which can be seen in Definition 2.4. Intuitively it says that the encryption of a message should not leak any

[1] To see this assume that both E and D are the identity functions, and hence ignoring k. Clearly that results in $D(k, E(k, m)) = m$ for all m, however everyone is able to efficiently compute m despite seeing its ciphertext form.

information. This is defined by giving an adversary $\mathcal{A}$ access to an encryption oracle, which will encrypt one of two messages he provides. If the adversary can guess which message is encrypted with non-negligible probability he has broken IND-CPA of the encryptions scheme.

DEFINITION 2.4 (SECRET-KEY IND-CPA) *The following game is used to define* IND-CPA *for a* (G, E, D) *symmetric-key encryption scheme, between an adversary* $\mathcal{A}$ *and challenger* $\mathcal{CH}$*:*
$\boldsymbol{IND\text{-}CPA}^b(\mathcal{A}, \kappa)$*:*

1. $\mathcal{CH}$ *generates* $k \leftarrow G(1^\kappa)$
2. $\mathcal{A}$ *gives two distinct messages* m_0, m_1 *to* $\mathcal{CH}$*, where* $|m_0| = |m_1|$
3. $\mathcal{CH}$ *returns* $E(k, m_b)$ *to* $\mathcal{A}$
4. $\mathcal{A}$ *outputs* b' *as result of the game*

The encryption scheme is secure against a chosen plaintext attack *if for any* κ *and any PPT adversary* $\mathcal{A}$ *his advantage is*

$$\left| Pr[\boldsymbol{IND\text{-}CPA}^0(\mathcal{A}, \kappa) = 1] - Pr[\boldsymbol{IND\text{-}CPA}^1(\mathcal{A}, \kappa) = 1] \right| \leq negl(\kappa)$$

A, usually impractical, symmetric encryption scheme is the one-time pad. It is based on having a long uniformly random key k, where the encryption algorithm XORs k and the plaintext, to obtain a ciphertext c. The decryption algorithm does the same, namely XORs k and c to obtain the plaintext. To ensure security the key is required to be at least the same length of the plaintext, and it can never be reused.

2.2.2 Partially homomorphic encryption scheme

A homomorphic encryption scheme is a public-key encryption scheme, which consists of four algorithms $(G, E, D, Eval)$. The algorithms are specified in Definition 2.5, where intuitively G outputs a pair of keys (pk, sk), where pk is used with E to encrypt messages, and sk is used with D to decrypt. The new algorithm $Eval$ is able to compute a function f on message m such that $f(m)$ is equivalent to encrypting m, using $Eval$ to evaluate f and then decrypting the resulting ciphertext. In a fully homomorphic encryption scheme f can be any function, however this thesis requires only a partially homomorphic encryption scheme, which means f is either addition or multiplication of plaintexts.

Definition 2.5 (Homomorphic Encryption Scheme) *A* homomorphic encryption scheme *is a tuple* $(G, E, D, Eval)$*, of PPT algorithms with the following properties:*

- *On input* 1^κ*, algorithm* G *(called the* key-generator*) outputs two bit strings* (pk, sk)*.*
- *For every string pair* (pk, sk) *in the range of* $G(1^\kappa)$*, and for every* $m \in \{0,1\}^*$*, algorithms* E *(encryption) and* D *(decryption) satisfy*

$$Pr[D(sk, E(pk, m)) = m] = 1$$

  where the probability is taken over the internal coin tosses of algorithms E *and* D

- *The* $Eval$ *algorithm satisfies, given function* f *and for every string pair* (pk, sk) *in the range of* $G(1^\kappa)$ *and vector of messages* $\boldsymbol{m}$*,*

$$D(sk, Eval(f, E(pk, \boldsymbol{m}))) = f(\boldsymbol{m})$$

This does not specify any sort of security, which is why two extra definitions will be used, namely IND-CPA, in Definition 2.6, and circuit-privacy, in Definition 2.7.

Intuitively Definition 2.6 says that no PPT algorithm allowed to chose two plaintexts m_0, m_1 can distinguish whether it obtained $E(pk, m_0)$ or $E(pk, m_1)$ when given $(pk, E(pk, m_b))$ for a uniformly random $b \in_R \{0,1\}$.

Definition 2.6 (Public-key IND-CPA, [Gol04]) *The following game is used to define* IND-CPA *for a* $(G, E, D, Eval)$ *public-key homomorphic encryption scheme, between an adversary* $\mathcal{A}$ *and challenger* $\mathcal{CH}$*:*
IND-CPA$^b(\mathcal{A}, \kappa)$*:*

1. $\mathcal{CH}$ *generates* $(pk, sk) \leftarrow G(1^\kappa)$*, and gives* pk *to* $\mathcal{A}$
2. $\mathcal{A}$ *gives two distinct messages* m_0, m_1 *to* $\mathcal{CH}$*, where* $|m_0| = |m_1|$
3. $\mathcal{CH}$ *returns* $E(pk, m_b)$ *to* $\mathcal{A}$
4. $\mathcal{A}$ *outputs* b' *as result of the game*

The encryption scheme is secure against a chosen plaintext attack *if for any* κ *and any PPT adversary* $\mathcal{A}$ *his advantage is*

$$\Big| Pr[\textbf{\textit{IND-CPA}}^0(\mathcal{A}, \kappa) = 1] - Pr[\textbf{\textit{IND-CPA}}^1(\mathcal{A}, \kappa) = 1] \Big| \leq negl(\kappa)$$

Circuit privacy is defined in Definition 2.7, and defines two probability distributions. The first consists of the secret key and a fresh encryption of $f(\boldsymbol{x})$, while the second consists of the secret key and $Eval$ used on f and the encryption of $\boldsymbol{x}$. It says that these two distributions are computationally indistinguishable, which means even given the secret key no PPT algorithm can distinguish be-

tween a fresh encryption of an evaluated function f from $Eval$ used on f and an encrypted message.

DEFINITION 2.7 (CIRCUIT PRIVACY, [ORL15]) *A $(G, E, D, Eval)$ public-key homomorphic encryption scheme satisfies* circuit privacy *if*

$$(sk, E(pk, f(\boldsymbol{m}))) \stackrel{c}{\equiv} (sk, Eval(f, E(pk, \boldsymbol{m})))$$

for any PPT distinguisher, where $(pk, sk) \to G(1^\kappa)$ and $\boldsymbol{m}$ is a vector of messages.

For completeness two partially homomorphic encryption schemes will be shown, namely Paillier's additive scheme [Pai99] and the multiplicative scheme based on ElGamal [Gam84]. For each scheme the four algorithms will be defined however the correctness and security properties are omitted, but can be seen in their respective papers.

Paillier's additively homomorphic encryption scheme

Paillier [Pai99] is a variant of RSA, and will be described using the simplified scheme from [Orl15]. Its key generation algorithm consists of finding two large primes p, q, and multiplying them to obtain the public key $p \cdot q = N$. The secret key is set to be $\phi(N) = (p-1) \cdot (q-1)$.

To encrypt message $m \in \mathbb{Z}_N$ the encryption algorithm E finds $r \in_{\mathrm{R}} \mathbb{Z}_N^*$ and computes the ciphertext

$$c = E(pk, m) = (1 + m \cdot N) \cdot (r^N) \mod N^2$$

To decrypt ciphertext c the decryption algorithm D computes

$$m = D(sk, c) = \frac{c^{\phi(N)} - 1 \mod N^2}{N} \cdot \phi(N)^{-1} \mod N$$

The $Eval$ algorithm computes the addition of two encrypted plaintexts m_0, m_1 by multiplying the two corresponding ciphertexts c_0, c_1, which means

$$Eval(+, (c_0, c_1)) = c_0 \cdot c_1 \mod N^2$$

and is denoted $c_0 +_H c_1$.

An important aspect of additive homomorphic encryption is being able to multiply an encrypted plaintext m by a constant w without knowing m. For [Pai99] this is done by computing c^w, where $c = E(pk, m)$, and is denoted $c * w$. This results in $D(sk, c * w) = m \cdot w \mod N$.

Multiplicative homomorphic encryption scheme based on ElGamal

[Gam84] works modulo p, where p is prime and another large prime q divides $p-1$. The group G is the subgroup Z_p^* which has order q and can be generated by a generator $g \in G$. The secret key consists of a uniformly random $x \in \{1, \ldots, q-1\}$. The public key is generated by computing the group element $h = g^x \mod p$, and then defining it as the tuple (G, p, q, g, h).

To encrypt message $m \in G$ the encryption algorithm finds a uniformly random $r \in \{1, \ldots, q-1\}$ and outputs the tuple

$$c = E(pk, m) = (g^r \mod p, m \cdot h^r \mod p)$$

The decryption algorithm is split into two steps, for ciphertext $c = (c_1, c_2)$. The first is computing

$$c_1^x = (g^r)^x = g^{x \cdot r} = s$$

and the second step inverts s and computes the message

$$m = c_2 \cdot s^{-1}$$

The *Eval* algorithm uses two ciphertexts $c, c' = (c_1, c_2), (c_1', c_2')$, and computes the multiplication of the group elements corresponding to their plaintexts by multiplying the ciphertexts:

$$c \cdot c' = (g^r, m \cdot h^r) \cdot (g^{r'}, m' \cdot h^{r'}) = (g^r \cdot g^{r'}, m \cdot h^r \cdot m \cdot h^{r'}) = (g^{r+r'}, m \cdot m' \cdot h^{r+r'})$$

which decrypts to $m \cdot m' \mod p$. Since both m and m' are contained in the group G so is the product of the m and m'.

2.3 Hashing and Bloom filters

This thesis will use three different data structures based on hash functions, namely simple hashing, Cuckoo hashing and Bloom filters. This section will first describe simple hashing, followed by showing the simplest version of Cuckoo hashing and then explain three optimizations to Cuckoo. Afterwards a basic Bloom filter will be explained, together with various modifications to obtain data structures related to Bloom filters. At last hashing failures will briefly be dealt with.

The two hashing schemes will be used to hash n items to a table T with b bins using h hash functions modelled as random functions, denoted as $\{H_1, \ldots, H_h\}$. Since only the insert function is relevant here the descriptions of lookups and deletes are omitted.

2.3.1 Simple Hashing

The basic idea of simple hashing is to store each element at all its hashed positions, resulting in $n \cdot h$ stored elements. In particular when hashing an element x it stores x in the bin $T[H_i(x)]$ for each $1 \leq i \leq h$. The bin size is not limited to 1, so to contain the multiple elements T will be denoted as a double array $T[\,][\,]$. The insert function, when inserting a single element, can be seen in Figure 2.3.

The maximum number of elements hashing to the same bin has been estimated in various papers, with different values for n, b and h. One estimate is made by [PSSZ15a], where $b = 2.4n$ and $h = 1$ for $2n$ balls (or $b = 2.4n$ and

```
1: procedure SimpleHashing.Insert(x)
2:     for i = 1 to h do
3:         if T[H_i(x)].length ≥ maxSize then return fail
4:         Insert x in the bucket T[H_i(x)]
```

Figure 2.3: Pseudocode for simple hashing insert algorithm, when inserting element x into double array T

```
1: procedure SimpleHashing.Insert(x)
2:     for i = 1 to h do
3:         Insert x in the bucket T[H_i(x)]
```

Figure 2.4: Pseudocode for simple hashing insert algorithm, when inserting element x into double array T of unlimited size

$h = 2$ for n balls). They wanted the chance of hashing more than k items to the same bin to be bounded by $\lambda = 2^{-40}$, which is achieved by setting k to be

$$k \geq \max(6, 2 \cdot e \cdot \log n / \log \log n)$$

A secondary version of Simple Hashing is also being used, where the bucket size is unlimited. This version can, for completeness, be seen in Figure 2.4.

2.3.2 Cuckoo Hashing

Cuckoo hashing, first proposed by [PR01], exists in various variants, and is a hashing scheme that stores at most one element in each bucket by relocating elements. The simplest variant uses 2 hash functions, where it stores n elements in $b = 2 \cdot (1 + \epsilon) \cdot n$ bins, for some small ϵ, with good probability. The basic insert function for a single element can be seen in Figure 2.5.

Intuitively an element x is hashed by first using H_1 to fix the position $pos = H_1(x)$. If the position is empty x can safely be stored there. Otherwise x and the element at $T[pos]$ are swapped, and pos is set to use the hash function that does not hash the new x to pos. The concept of looping with different elements is done at most $size$ times, where $size$ is the number of elements currently in the array, at which points one of two things will have happened. Either an empty position has been found and the algorithm has terminated, which will happen with good probability, or it has not found an empty position, in which case two new hash functions will be sampled and all elements will be rehashed.

Various modifications have been made to Cuckoo hashing, where three are of particular interest here.

Including a stash In the basic form Cuckoo hashing suffers from having a small but not negligible probability of failing, if some subset of t elements all hash to the same $t - 1$ positions for both hash functions, called a cycle. [KMW09] proposes to use a constant size stash to avoid this issue, by evicting one of the t elements and inserting it into the stash, and thus being able to

```
1: procedure CuckooHashing.Insert(x)
2:     if T[H_1(x)] = x or T[H_2(x)] = x then return
3:     pos ← H_1(x)
4:     for i = 0 to size do
5:         if T[pos] = ⊥ then T[pos] ← x; return
6:         x ↔ T[pos]
7:         if pos = H_1(x) then pos ← H_2(x)
8:         else pos ← H_1(x)
9:     rehash(); insert(x)
```

Figure 2.5: Pseudocode for the basic Cuckoo hashing insert algorithm, when inserting element x into array T with 2 hash functions, [PR01]

```
1: procedure CuckooHashing.Insert(x)
2:     if T[H_1(x)] = x or T[H_2(x)] = x then return
3:     pos ← H_1(x)
4:     for i = 0 to size do
5:         if T[pos] = ⊥ then T[pos] ← x; return
6:         x ↔ T[pos]
7:         if pos = H_1(x) then pos ← H_2(x)
8:         else pos ← H_1(x)
9:     if |stash| < s then stash.add(x); return
10:    rehash(); insert(x)
```

Figure 2.6: Pseudocode for Cuckoo hashing insert algorithm with stash, when inserting element x into array T with 2 hash functions and stash of size s, [KMW09]

hash the remaining $t-1$ elements into the $t-1$ buckets. This approach avoids rehashing all the elements with new hash functions, at the cost of maintaining a small stash. Experimental results by [PSSZ15a] concluded that a stash of size 4 to 6 reduces the chance of rehashing to almost zero, when hashing between 2^{12} and 2^{24} elements.

The pseudocode of the modified algorithm can be seen in Figure 2.6, where the changes have been highlighted in bold, and the stash size is denoted s.

Multiple hash functions A second modification is using more than two hash functions, which reduces the number of required bins, and thus getting a better space utilization. This is denoted h-ary Cuckoo hashing, for h hash functions. Experimental results from [FPSS03] obtains 91% space utilization with three hash functions and 97% with four hash functions compared to the 50% for two hash functions.

The pseudocode of the modified algorithm can be seen in Figure 2.7, where the changes have not been highlighted due to most of the pseudocode being different.

```
1:  procedure CuckooHashing.Insert(x)
2:      Let set H ← {H_1, ..., H_h}
3:      if T[H_j(x)] = x for some H_j ∈ H then return
4:      for i = 0 to size do
5:          for H_j ∈ H do
6:              if T[H_j(x)] = ⊥ then T[H_j(x)] ← x; return
7:          Pick random H_j ∈ H
8:          Swap x and T[H_j(x)]
9:          Set H ← {H_1, ..., H_h} \ H_j
10:     if |stash| < s then stash.add(x); return
11:     rehash(); insert(x)
```

Figure 2.7: Pseudocode for Cuckoo hashing insert algorithm with stash, when inserting element x into array T with h hash functions and stash of size s, [FPSS03]

Cuckoo hashing with two hash functions is able to determine which hash function to use, given an element and a position. With three or more hash functions there are at least two hash functions that can be used to continue, which has led to different algorithms that decide which hash function to proceed with. Here the simplest scheme will be shown, which means picking a random one from the remaining $h - 1$.

The number of bins b required to hash n elements using h hash functions will be denoted $b = F'(n, h)$, as the required number of bins is dependant on which insertion algorithm is used.

Reducing length of stored elements The last optimization for Cuckoo hashing is made by [ANS10]. The paper presents a construction that reduces the bit-length of the stored elements as a way to significantly reduce the number of bits required to represent the whole cuckoo hash table. [PSSZ15a] uses a variant that uses slightly more space, while still using the same structure as the previously mentioned Cuckoo hashing algorithms.

By using h random functions modelled as random oracles, $f_1, \ldots, f_h$, with the range $[0, b-1]$ an element $x = x_L \parallel x_R$ has x_R mapped to bin $x_L \oplus f(x_R)$. As $|x_L| = \log(b)$ this means the stored elements have their lengths reduced by $\log(b)$, assuming a random function f that is k-wise independent, where $k = polylog(\kappa)$.

The correctness of this scheme can be seen if two elements $x_L \parallel x_R$ and $x'_L \parallel x_R$ are hashed to the same bin with the same function while having equal suffix. This must mean that $x_L \oplus f(x_R) = x'_L \oplus f(x_R)$. Since $f(x_R)$ can be removed $x_L = x'_L$ which means that it is the same element.

The pseudocode can be seen in Figure 2.8. Compared to the previous Cuckoo hashing insert algorithms the hash functions have been replaced by the random functions.

Of note is Line 11 which assumes it is possible, given x_R and an entry

```
1: procedure CuckooHashing.Insert(x)
2:     Split x into x = x_L || x_R where |x_L| = log b
3:     Let set F = {f_1, ..., f_h}
4:     if T[x_L ⊕ f_j(x_R)] = x_R for some f_j ∈ F then return
5:         for i = 0 to size do
6:             for f_j ∈ F do
7:                 if T[x_L ⊕ f_j(x_R)] = ⊥ then
8:                     T[x_L ⊕ f_j(x_R)] ← x_R; return
9:             Pick random f_j ∈ F
10:            Swap x_R and T[x_L ⊕ f_j(x_R)]
11:            Compute x = F^{-1}(x_R)
12:            Set F ← {f_1, ..., f_h} \ f_j
13:        if |stash| < s then stash.add(x); return
14:        rehash(); insert(x)
```

Figure 2.8: Pseudocode for Cuckoo hashing insert algorithm with stash, when inserting element x into array T with h random functions and stash of size s, [ANS10]

e, to find x. This can be done by calculating $x_L = f_j(x_R) \oplus e$. Since it is not immediately obvious which random function applies to a given element a secondary array can be stored to remember which of the h random function has been used for each bin, at the cost of using $b \cdot \log(h)$ extra bits. In the same manner Lines 4, 7 and 8 assume this mapping as well. This notation has been omitted from Figure 2.8 to ease notation, and Chapter 4 will elaborate on this point.

2.3.3 Bloom Filter

[Blo70] proposes Bloom filters (BF) as a data structure for testing set membership. A (m, n, h, H)-Bloom filter consists of m buckets using one bit each, to represent n elements with hash functions $H = \{H_1, \ldots, H_h\}$, where each hash function maps to $[0; m-1]$.

An initially empty Bloom filter will have all m bits equal to 0. To insert an element x the h hash functions are evaluated to set $H_i(x) = 1$ for all $i \leq k$. The pseudocode for insertion can be seen in Figure 2.9. Checking whether some element x is contained in the set represented by a Bloom filter can be done by verifying whether $H_i(x) = 1$ for all i. If there exists a position where $H_i(x) = 0$ it is guaranteed to not be in the set, and if all positions are 1 then it is very likely to be contained in the set. This can also be stated as there are no false negatives and the chance of false positive is negligible, with properly chosen parameters.

2.3.4 Inverted and Encrypted Bloom filter

[DC16] uses two small changes to Bloom filters, namely inverting and encrypting

```
1: procedure BloomFilter.Insert(x)
2:     for i = 1 to h do
3:         BF[H_i(x)] ← 1
```

Figure 2.9: Insertion algorithm for Bloom filter, when inserting element x with h hash functions $H = \{H_1, \ldots, H_h\}$, [Blo70]

them. An inverted Bloom filter IBF made from BF will have the following entries:

$$IBF[i] = \begin{cases} 0 & \text{if } BF[i] = 1 \\ 1 & \text{otherwise} \end{cases}$$

[DC16] also uses another variant of Bloom filters, the Encrypted Bloom filters (EBF). This is closely related to ordinary Bloom filters, as the insert algorithm looks exactly like Figure 2.9. After all elements have been inserted the entries will be encrypted using a homomorphic encryption key pk, to obtain the Encrypted Bloom filter

$$EBF[i] = E(pk, BF[i])$$

for each $1 \leq i \leq m$.

At last [DC16] also combines them, to obtain an encrypted inverted Bloom filter (EIBF), which is a Bloom filter that has been inverted and then encrypted.

2.3.5 Garbled Bloom Filter

[DCW13] introduces a variant of Bloom filters called garbled Bloom filters (GBF). The primary difference lies in using κ bits to represent each of the m entries, compared to the one bit Bloom filters use. The κ bits are used to hide whether an element is hashed to the position, by using the XOR-based secret sharing scheme to insert uniformly random κ-bit values. The new parameterization is (m, n, h, H, κ)-garbled Bloom filter, where m, n, h and H are defined as previously and κ is the security parameter.

The pseudocode when inserting an element can be seen in Figure 2.10.

Intuitively it manages two variables, $finalShare \leftarrow x$ and $emptySlot \leftarrow -1$. For each hash function H_i the algorithm finds $e = GBF[H_i(x)]$, and one of three things happen:

- If e is equal to $\perp$ and $emptySlot = -1$ it updates $emptySlot$ to be position $H_i(x)$
- If e is equal to $\perp$ and $emptySlot \neq -1$ a uniformly random value is inserted into position $H_i(x)$ and $finalShare$ is XORed with the newly inserted value: $finalShare \leftarrow finalShare \oplus GBF[H_i(x)]$
- If e is not equal to $\perp$ it updates $finalShare$ as before, namely by computing $finalShare \leftarrow finalShare \oplus e$

```
1: procedure GarbledBloomFilter.Insert(x)
2:     emptySlot ← −1, finalShare ← x
3:     for i = 1 to h do
4:         j ← H_i(x)
5:         if GBF[j] = ⊥ then
6:             if emptySlot = −1 then
7:                 emptySlot ← j
8:             else
9:                 GBF[j] ∈_R {0,1}^κ
10:                finalShare ← finalShare ⊕ GBF[j]
11:        else
12:            finalShare ← finalShare ⊕ GBF[j]
13:    GBF[emptySlot] ← finalShare
```

Figure 2.10: Insertion algorithm for garbled Bloom filter, when inserting element x with h hash functions $H = \{H_1, \ldots, H_h\}$, copied with modifications from [DCW13]

After all hash functions have been used $finalShare$ is inserted into the position $emptySlot$, which results in the equation

$$x = \bigoplus_{i=1}^{h} GBF[H_i(x)]$$

being true for all inserted elements.

When all n elements have been inserted a final GBF is produced by inserting uniformly random κ-bit values into all empty positions, thus making it impossible for a PPT algorithm to distinguish whether the GBF has any elements hashed to a given position or not, given one entry. To reduce the chance of failures to be negligible [DCW13] uses $h = \kappa$ hash functions, and sets $m = \log(e) \cdot h \cdot n \approx 1.44 \cdot \kappa \cdot n$. Proofs of correctness for GBF can be seen in [DCW13].

[DCW13] proposes to use κ hash functions (and κ-bit long entries) for their data structure to ensure the probability that anyone obtaining half of the garbled Bloom filter will learn nothing, except with negligible probability, or carefully chosen indices which will be described in Chapter 5. To see this assume an adversary is given $m/2$ random entries of a garbled Bloom filter GBF with m entries. To check whether some element x is contained in the GBF he needs to get all entries at $GBF[H_k(x)]$, for $1 \leq k \leq \kappa$, to verify whether the entries XOR to x. The probability he has any one of them is $1/2$, and extending it to all hash functions results in $2^{-\kappa}$ probability of not missing any entries for x, which is negligible, and thus the adversary learns nothing.

2.3.6 A note on hashing failures

The constructions of both hashing schemes and the Garbled Bloom filters have a chance of failing. The simple hashing scheme can fail if a bin is overpopulated[2], which in particular means that more than t elements are hashed to a bin, which can contain at most t elements.

The Cuckoo hashing scheme can fail if (with no stash) there is a cycle, which means that t elements hash to the same $t-1$ spots, or, with a stash of size s, if there are more than s cycles.

The garbled Bloom filters can fail in two scenarios. First if some element hashes to taken positions for each of the κ hash functions which might be possible to fix by rehashing all elements, starting with the failed one. The second, and more extreme, is if two different elements x_0, x_1 hash to the same positions for all hash functions. This means it is impossible to satisfy

$$x_b = \bigoplus_{i=0}^{\kappa} GBF[H_i(x_b)]$$

for both elements, as that would imply

$$x_0 = \bigoplus_{i=0}^{\kappa} GBF[H_i(x_0)] = \bigoplus_{i=0}^{\kappa} GBF[H_i(x_1)] = x_1$$

A protocol that hashes using public or shared hash functions has to handle these failures, as it cannot trivially change hash functions without leaking information. Outside of reducing the chance of failures to be negligible [PSZ14] mentions two ways of handling this, either by requesting new hash functions or calculate an approximation of the functionality, which means leaving out elements that cause the hashing to fail and thus having at most statistical correctness. Note that the correctness issue here means removing potential elements, rather than including false positives.

The first strategy leaks that the party's input cannot be properly hashed with the shared hash functions, which makes it possible to check whether a given set is the other party's input. The second possibility can leak information if the protocol is run multiple times, since these approximations might be detected.

Deciding which strategy is superior must be checked on a case-by-case basis.

2.4 Security definitions

This section will primarily focus on the definitions of a secure protocol, and how to prove a given protocol is secure. However first some variants of adversaries will be explained, followed by the description of what a functionality is.

Adversarial powers: Cryptography models adversarial powers by defining different variables, where the most relevant ones will be outlined here.

[2] if the bucket size is limited, otherwise it clearly cannot become overpopulated.

Usually two types of adversaries are considered, to model the behaviour of a corrupt party. The first case is the semi-honest adversary, sometimes mentioned as passive or honest-but-curious. A semi-honest adversary will complete a protocol as it is specified, while trying to obtain extra information from the received messages. Secondly a malicious, or active, adversary is allowed to behave arbitrarily.

Protocols that are secure against a malicious adversary clearly provide a stronger sense of security, however the notion of semi-honest adversaries is relevant for various reasons. One reason is the efficiency of semi-honestly secure protocols, which usually is significantly higher than their equivalent maliciously secure protocols.

In some cases sophisticated software can force the protocol to be used in a black-box manner, making it impossible for a user to choose inputs, or in some other way is forced to behave correctly. Alternatively a leaked transcript of messages forces the adversary to behave nicely, as he cannot influence the protocol.

Another feature is the time of corruption, which can be either adaptive or static. In the adaptive case an adversary can choose to corrupt either party at any time during the execution of a protocol, while a static adversary will corrupt a party prior to the protocol starting. The work done here only considers static adversaries.

As the focus is on two-party protocols the threshold number of parties that will be corrupted is at most one, which means either the Sender, the Receiver or neither will be corrupted.

A last distinction is the computing power of an adversary. A computationally bounded adversary is allowed to compute any PPT algorithm, while an unconditional adversary can spend as much time as he requires. The focus here will be on computationally bounded adversaries.

Functionality: A protocol consists of four parts, namely the input, auxiliary input, a random-tape and the received messages, from which the output can be calculated. To prove that a protocol is secure reduces to showing that the received message leak no unintended information. To do this a comparison between the protocol and something that is secure by definition, a functionality, is made.

A functionality can be viewed as a trusted third party calculating a defined function. This is done by having both parties send their inputs' to the functionality, which calculates the outputs and returns them, which is described as $f(x, y) = (f_1(x, y), f_2(x, y))$. A functionality only outputs the result of the computation, meaning it cannot leak any unintended information, and hence is secure per definition.

2.4.1 Semi-Honest adversary

A formal definition of security against semi-honest adversaries in two party protocols was shown by [HL10], which can be seen in Definition 2.8. It is defined for parties P_1 and P_2 with, respectively, input x and y. They wish to

perform the protocol π which computes the PPT functionality $f = (f_1, f_2)$. The used notation is as follows:

$view_i$, for $i \in \{1, 2\}$, is the view of the i'th party during π when computed with inputs x, y and security parameter κ. The view is presented as a tuple $(w, r^i, m_1^i, \ldots, m_t^i)$ where w is the input (x or y), r^i is the random tape and $(m_1^i, \ldots, m_t^i)$ are the t messages P_i receives.

$output_i^\pi(x, y, \kappa)$ denotes the output of party i when running protocol π on inputs x and y with security parameter κ. The joint output of both parties is denoted $output^\pi(x, y, \kappa) = (output_1^\pi(x, y, \kappa), output_2^\pi(x, y, \kappa))$. Note that the output of a party can be computed from its view.

DEFINITION 2.8 (SECURITY - SEMI-HONEST FORMULATION, [HL10])
Let $f = (f_1, f_2)$ be a deterministic functionality. We say that π securely computes *f in the presence of a static semi-honest adversary if*

$$\{output^\pi(x, y, \kappa)\}_{x,y\in\{0,1\}^*;\kappa\in\mathbb{N}} \stackrel{c}{\equiv} \{f(x, y)\}_{x,y\in\{0,1\}^*} \quad (2.1)$$

and there exists PPT algorithms S_1, S_2 such that

$$\{S_1(1^\kappa, x, f_1(x, y))\}_{x,y\in\{0,1\}^*,\kappa\in\mathbb{N}} \stackrel{c}{\equiv} \{view_1^\pi(x, y, \kappa)\}_{x,y\in\{0,1\}^*,\kappa\in\mathbb{N}},$$
$$\{S_2(1^\kappa, y, f_2(x, y))\}_{x,y\in\{0,1\}^*,\kappa\in\mathbb{N}} \stackrel{c}{\equiv} \{view_2^\pi(x, y, \kappa)\}_{x,y\in\{0,1\}^*,\kappa\in\mathbb{N}} \quad (2.2)$$

Intuitively Definition 2.8 requires two separate things to be proven. First Formula 2.1 says the output of the functionality has to be computationally indistinguishable from the output of the protocol, on the same inputs. This in practice means that the outputs have to be equivalent, except with negligible probability.

Secondly Formula 2.2 requires two simulators to be constructed. For party i the simulator S_i needs to, given P_i's input and output, construct a view of the protocol that is computationally indistinguishable from the view when computing the protocol. This means simulating the messages received from the other party so that it cannot be distinguished from the real protocol, while being consistent with the input and output.

2.4.2 Malicious adversary

To model the behaviour of a party that is allowed to behave arbitrarily there are three weaknesses that need to be addressed, when comparing it to the semi-honest adversary.

First of all a party can refuse to participate, leading to a very short protocol without a correct output. Secondly a malicious party is not guaranteed to use his predefined input, meaning simulating a protocol given the input and output is no longer valid, as the input might not even be well-defined. Lastly a malicious party can refuse participating in the protocol at any given point, leaving an honest party with a partially finished protocol. This in general means

EXECUTION IN THE IDEAL MODEL

1. Inputs: Let x denote the input of P_1, y the input of P_2. Let adversary $\mathcal{A}$ get auxiliary input denoted z.

2. Send inputs to trusted party: The honest party P_j sends its received input to the trusted party. The corrupted party P_i controlled by $\mathcal{A}$ may either abort (by replacing the input with a special $abort_i$ message), send its received input, or send some other input of the same length to the trusted party. This decision is made by $\mathcal{A}$ and may depend on the input value of P_i and the auxiliary input z. Denote the pair of inputs sent to the trusted party by (x', y').

3. Early abort option: If the trusted party receives an input of the form $abort_i$ for some $i \in \{1, 2\}$, it sends $abort_i$ to all parties and the ideal execution terminates. Otherwise, the execution proceeds to the next step.

4. Trusted party sends output to adversary: At this point the trusted party computes $f_1(x', y')$ and $f_2(x', y')$ and sends $f_i(x', y')$ to party P_i, which is the corrupt party.

5. Adversary instructs trusted party to continue or halt: $\mathcal{A}$ sends either *continue* or $abort_i$ to the trusted party. If it sends *continue*, the trusted party sends $f_j(x', y')$ to party P_j, the honest party. Otherwise, if $\mathcal{A}$ sends $abort_i$, the trusted party sends $abort_i$ to party P_j.

6. Outputs: The honest party always outputs the output value it obtained from the trusted party. The corrupted party outputs nothing. The adversary $\mathcal{A}$ outputs any arbitrary (PPT computable) function of the initial input of the corrupted party, the auxiliary input z, and the value $f_i(x', y')$ obtained from the trusted party.

Figure 2.11: Functionality $\mathcal{F}^f$, copied almost verbatim from [HL10]

that a two-party protocol is never guaranteed fairness, as there is no mechanism to prevent a party from stopping.

[HL10] defines security with a statically corrupted malicious adversary by comparing what he can do in the real world to what he is capable of in the ideal model.

The ideal model is specified by having a trusted third party calculate the functionality without revealing anything, leading it to per definition be secure. For a functionality $f : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^* \times \{0,1\}^*$ the ideal execution can be seen in Figure 2.11.

Intuitively Figure 2.11 models an ideal world where the three weaknesses are integrated as parts of it. This can be seen in the second step, where a corrupt P_i is allowed to abort the protocol or change his input. An important note about P_i's input to the trusted party is that it cannot depend on P_j's input, but it can depend on any other arbitrary decision. In the fourth step $\mathcal{A}$ obtains his output, and afterwards he is allowed to stop the protocol, which models the lack of fairness with a malicious adversary.

A formal definition of how the ideal world has to be able to simulate the

real world can be seen in Definition 2.9.

$IDEAL_{f,\mathcal{A}(z),i}(x, y, \kappa)$ is referred to as the execution of the ideal model on functionality f with adversary $\mathcal{A}$ on auxiliary input z, where party P_i is corrupt, on the inputs are x and y and security parameter κ.

$REAL_{\pi,\mathcal{A}(z),i}(x, y, \kappa)$ is the real execution of protocol π with adversary $\mathcal{A}$ on auxiliary input z, where party P_i is corrupt, on the inputs x and y and security parameter κ.

DEFINITION 2.9 (SECURITY - MALICIOUS FORMULATION, [HL10]) *Let f be as in Figure 2.11 and π be a two-party protocol for computing f. Protocol π is said to* securely compute *f* in the presence of malicious adversaries *if for every non-uniform PPT adversary $\mathcal{A}$ for the real model, there exists a non-uniform PPT adversary $\mathcal{S}$ for the ideal model, such that for every $i \in \{1, 2\}$,*

$$\{IDEAL_{f,\mathcal{S}(z),i}(x, y, \kappa)\}_{x,y,z,\kappa} \stackrel{c}{\equiv} \{REAL_{\pi,\mathcal{A}(z),i}(x, y, \kappa)\}_{x,y,z,\kappa}$$

where $x, y \in \{0,1\}^*$ under the constraint that $|x| = |y|, z \in \{0,1\}^*$ and $\kappa \in \mathbb{N}$.

Intuitively the definition says that for any strategy an adversary uses in the real world there exists a strategy in the ideal world where the adversary obtains the same information. Since the ideal world is secure by definition the strategy is not an attack on the protocol.

2.4.3 Composability

An important aspect of cryptography protocols is composability, which means that a protocol proven secure in some environment can be used as a primitive for other protocols in the same environment. This includes reusing a specific subprotocol, or using multiple protocols in succession.

There are two reasons composability is important. First proving that a protocol is composable provides security in itself, since that means running some protocol multiple times in sequence does not weaken security - in fact that is exactly as secure as running it once. Secondly it is possible to create complicated protocols using various subprotocols. When proving security for such a construction every subprotocol can be proven secure by itself, and the primary protocol can be proven secure where each subprotocol is abstracted away, by replacing them with a trusted party providing the resulting computation, and thus significantly reducing the complexity of the proof. This is often referenced as the hybrid world.

In [Gol04] composability is proven for both semi-honest and maliciously secure protocols, and will thus not be included. The theorems have for completeness been included, see Theorem 2.10 and 2.11.

Theorem 2.10 (Composition Theorem - Semi-honest, [Gol04])
Suppose that g is privately reducible to f and that there exists a protocol for privately computing f. Then there exists a protocol for privately computing g.

Theorem 2.11 (Composition Theorem - Malicious, [Gol04])
Suppose that g is securely reducible to f and that there exists a protocol for securely computing f. Then there exists a protocol for securely computing g.

2.5 Oblivious Transfer

Oblivious Transfer (OT) is a cryptographic primitive which is a two-party protocol consisting of a Sender and a Receiver. In its most basic form the Sender's input consists of two values x_0, x_1, and the Receiver's input is a single bit b. An OT-protocol then aims to implement the functionality

$$f_{OT}((x_0, x_1), b) = (\perp, x_b)$$

without revealing neither x_{1-b} to the Receiver, nor b to the Sender. This version of OT is denoted 1-out-of-2 OT or $\binom{2}{1}$-OT, where a generalization is t-out-of-N OT, denoted $\binom{N}{t}$-OT, where the Sender has N values and the Receiver has to learn t of them.

The efficiency of OT protocols has been researched extensively, as performing a large number of basic OTs is expensive. [IKNP03] showed one of the largest improvements to the running-time of OTs, by presenting a general construction which extends a few expensive OTs to efficiently obtain many cheap OTs. Their construction is called OT-extensions and is secure in the random oracle model against a semi-honest adversary.

[KK13] proposes two modifications to OT-extensions. Their first result was a new construction called Rand-$\binom{N}{1}$-OT_k^i. This is an OT protocol resulting in N random strings to the Sender, and the Receiver is able to choose 1 of these strings. The strings are of length k, and the protocol is repeated i times. Compared to [IKNP03] this construction performs fewer computations and sends less messages. Their second improvement reduces the running-time of both the basic OT-extension and their random OT-extension, where both of their constructions are secure in the random oracle model, against a semi-honest adversary.

[ALSZ15] proposes a protocol that is secure in a malicious setting. Their resulting OT is a $\binom{2}{1}$-OT, which is only slightly less efficient than what was shown in [IKNP03], while using the optimization from [KK13].

To show the efficiency they implemented and compared various OT-extension constructions, leading to the Table 2.1, where only the most relevant running times have been included here. The results are with 128-bit security, are averaged over 2^{26} runs and use the notation $1KB = 8.192\text{bit}$. Their results show only a slight decrease in performance to obtain security against malicious adversaries.

Prot	Security	Run-Time		Communication	
		Local	**Cloud**	**Local**	**Cloud**
[IKNP03]	Passive	$0.3s + 1.07\mu s \cdot t$	$0.7s + 4.24\mu s \cdot t$	$4KB + 128bit \cdot t$	
[ALSZ15]	Active	$0.7s + 1.29\mu s \cdot t$	$1.3s + 6.92\mu s \cdot t$	$24KB + 191bit \cdot t$	$22KB + 175bit \cdot t$

Table 2.1: Table 1 taken almost verbatim from [ALSZ15]. Run times of different OT extensions.

2.6 Private Set Intersection

Before analyzing the four private set intersection protocols the usefulness of PSI protocols and its definition will be provided, followed by some related work.

Efficient PSI protocols have various real-world applications, and thus provide an interesting study. Example include, but are not limited to:

- Comparing a list of known terrorists to a foreign flight's passenger manifest. If both lists are confidential PSI protocols can be used to avoid breaking privacy laws, while ensuring no known terrorists are being transported.
- Companies that have signed exclusive deals with some number of customers. Without breaking confidentiality these lists can be compared between companies, to avoid dishonest customers.
- Multiple hospitals that want to use their private patient data for medical research, without breaking confidentiality.

The PSI protocols analyzed in this thesis consist of two parties, the Sender and the Receiver, where the Sender's input consists of the set $X = \{x_1, \ldots, x_{n_1}\}$ and the Receiver's input is $Y = \{y_1, \ldots, y_{n_2}\}$. The functionality being implemented is $f_{PSI}(X, Y) = (\perp, X \cap Y)$, meaning the Sender outputs nothing and the Receiver outputs the intersection. Usually both set sizes are leaked, resulting in the Receiver learning n_1 and the Sender learning n_2.

At the end of a PSI protocols the Receiver knows n_1 and $X \cap Y$, from which he can compute $|X \setminus Y| = |X \cap Y| - |n_1|$, which means he knows the number of elements in the Sender's set that is not contained in the intersection. What he does not learn is which elements, meaning excluding the elements in Y, the Receiver is not supposed to learn whether any other elements are contained in X.

Various types of PSI protocols have been defined, outside of the standard two-party case sketched above. One such type is the server-aided PSI, which is based on having two or more computationally weak parties, and a server which has significantly better hardware. [KMRS14] shows two protocols of this type. The first is a semi-honestly secure protocol for n parties and one server, where the server is assumed to not collude with any of the parties. The second is a maliciously secure two-party PSI protocol. Both are based on using pseudorandom permutations to hide the parties inputs, after which the server computes the intersection and sends it to the parties.

A third type of PSI protocols is Authenticated PSI (APSI) which uses a somewhat trusted third party to verify and commit to the input of some party. This notion has been used in [CT10], which based on RSA and the trusted party computes PSI in linear time, where the Receiver is allowed to be malicious while the Sender and trusted third party have to be semi-honest.

APSI is also used in [DC16] to modify its semi-honestly secure protocol to be maliciously secure. The semi-honestly secure protocol will be analyzed as part of this thesis, while the maliciously secure protocol is omitted.

A fourth way to compute PSI is using generic protocols like Yao's garbled circuits [Yao86] and optimize them for PSI. [HEK12] proposes three protocols to compute PSI, where the third protocol has an overhead of $\Theta(n \cdot \log(n))$ in the semi-honest model. Despite not being linear [HEK12] reports fairly competitive running times during experimental evaluation.

Chapter 3

OT-based PSI by [PSZ14]

In this chapter one method to compute PSI based on random OTs will be explained. Due to the complexity of the protocol this will be done through a series of smaller protocols, where at the end the full OT-based PSI protocol from [PSZ14] will be explained. At the end a proof of security will be given, as it was omitted from their paper.

3.1 Intuition behind OT-based PSI

The exact mechanics in the protocol of [PSZ14] are complicated so the basic idea behind the protocol will be thoroughly explained through a series of protocols that are secure against a static semi-honest adversary, where each protocol partially compute private set intersection. As their purpose purely is to explain OT-based PSI no proofs will be included, though an argument for correctness and security will be included.

3.1.1 Equality test

In a basic equality test two parties, the Sender and the Receiver, want to know if their input elements x and y, with bit-length σ, are equivalent, which means computing the functionality $f(x, y) = (\perp, x \stackrel{?}{=} y)$. The final version of the equality test can be seen in Figure 3.1.

The functionality can be computed by using σ Rand-$\binom{2}{1}$-OTs to create two masks that can be used to compare the elements. Both parties will split up their element into σ 1-bit pieces, denoted $x = x[1] \,\|\, \cdots \,\|\, x[\sigma]$ and $y = y[1] \,\|\, \cdots \,\|\, y[\sigma]$. For each $1 \leq k \leq \sigma$ OTs the Sender will obtain two random strings (m_0^k, m_1^k), while the Receiver will input the bit $y[k]$ to the OT and obtain the random string $m_{y[k]}^k$. At the end the Sender and Receiver will, respectively, compute

$$m_x = \bigoplus_{k=1}^{\sigma} m_{x[k]}^k \text{ and } m_y = \bigoplus_{k=1}^{\sigma} m_{y[k]}^k$$

The Sender then sends m_x to the Receiver, who will output $m_x \stackrel{?}{=} m_y$.

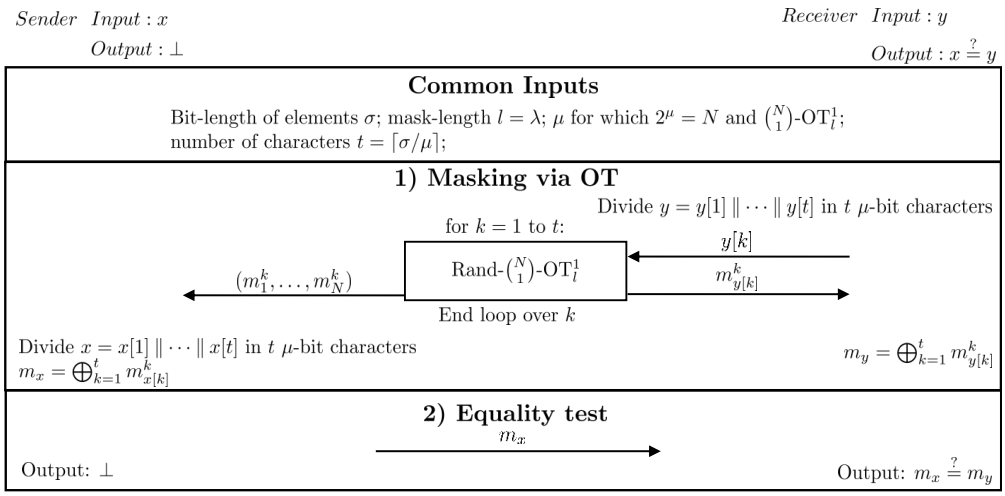

Figure 3.1: Semi-honestly secure equality test protocol

Clearly if $x = y$ it follows that

$$m_x = \bigoplus_{k=1}^{\sigma} m^k_{x[k]} = \bigoplus_{k=1}^{\sigma} m^k_{y[k]} = m_y$$

If $x \neq y$ the chance of a collision has to be negligible, which is achieved by having the OT output strings of length λ, the statistical security parameter. If x differs from y there is going to be at least one bit which is different, which results in the computation of m_x and m_y to XOR different random masks. It follows that the probability of m_x being equal to m_y is $2^{-\lambda}$, which means the chance of a false positive is negligible in the statistical security parameter.

The protocol hides the inputs of both parties. Clearly the Sender cannot learn y, as he does not obtain any messages from the Receiver, and the Receiver only learns x in the case where $x = y$, otherwise the mask m_x will be a uniformly random value leaking no information about x.

The equality test can be optimized by reducing the number of OTs, which means using a Rand-$\binom{N}{1}$-OT$^t_\lambda$, where $t = \lceil \sigma / \log(N) \rceil$. This means x and y will be split into t blocks of size μ, where $2^\mu = N$ (or $\mu = \lceil \log(N) \rceil$) that can be used as input to the OT. In particular x will be split into t μ-bit pieces which will be denoted $x = x[1] \parallel \cdots \parallel x[t]$. The OT output is now an N-tuple $(m^k_1, \ldots, m^k_N)$ and the generated mask is computed as $m_x = \bigoplus_{k=1}^t m^k_{x[k]}$. Equivalent steps are done to split y and generate m_y. This optimization follows as a result of [KK13] which proposes a Rand-$\binom{N}{1}$-OT which outperforms doing $\log(N)$ Rand-$\binom{2}{1}$-OT, when N is not too large.

The running time for an equality test is, assuming constant-time OT, only dependant on the length of the elements.

3.1.2 Inclusion test

The same concept of generating masks can be extended to test inclusion of an element in a set. The Sender has a set $X = \{x_1, \ldots, x_{n_1}\}$ and the Receiver

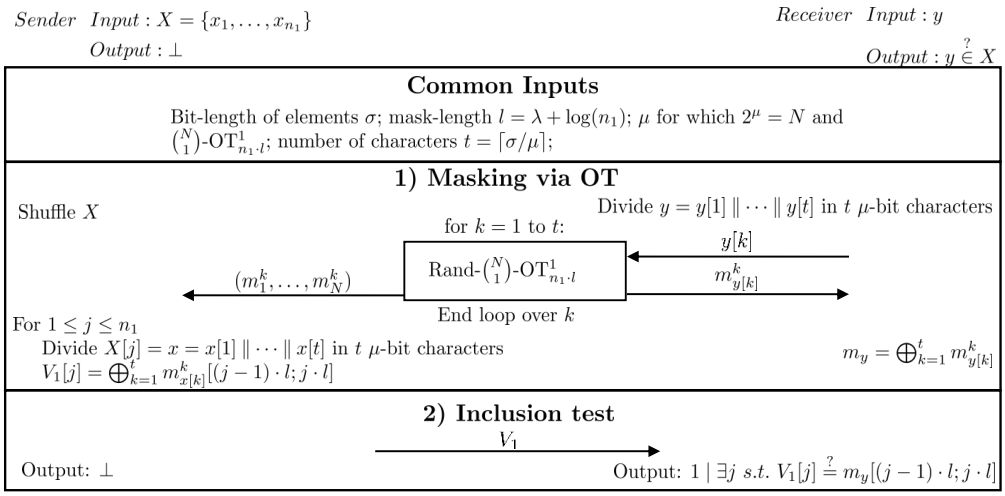

Figure 3.2: Semi-honestly secure inclusion test protocol

has an element y, where the two parties want to compute the functionality $f(X, y) = (\perp, y \overset{?}{\in} X)$. The inclusion test protocol can be seen in Figure 3.2.

Extending the equality test concept to testing inclusion can be done by increasing the length of the values output by the OT, and only using parts of the longer output to generate a given mask. The OT output length will be increased from λ to $n_1 \cdot l$, where l will be defined shortly, which results in using a Rand-$\binom{N}{1}$-OT$^t_{n_1 \cdot l}$, for $t = \lceil \sigma / \log(N) \rceil$. The algorithm for testing inclusion is as follows:

The Sender will at first shuffle his set X, followed by both parties engaging in t Rand-$\binom{N}{1}$-OT$_{n_1 \cdot l}$, where the Sender will obtain the output strings $(m_1^k, \ldots, m_N^k)$ and the Receiver will obtain the string $m^k_{y[k]}$, for $1 \leq k \leq t$. The Sender will compute a mask array V_1 as[1]

$$V_1[j] = \bigoplus_{k=1}^{t} m^k_{x[k]}[(j-1) \cdot l; j \cdot l]$$

for the j'th element, where V_1 has size n_1. The Sender will then send V_1 to the Receiver who will compute the XOR of all his strings and then check if

$$V_1[j] = m_y[(j-1) \cdot l; j \cdot l] \text{ for } 1 \leq j \leq n_1$$

If a match is found y is contained in X, except with negligible probability.

During the protocol n_1 comparisons will be computed, which means that the element length has to be increased, to ensure a negligible chance of failure. This results in $l = \lambda + \log(n_1)$-bit elements.

The protocol succeeds due to, assuming y is equal to the j'th element in X after shuffling,

$$\bigoplus_{k=1}^{t} m^k_{y[k]}[(j-1) \cdot l; j \cdot l] = m_y[(j-1) \cdot l; j \cdot l] = V_1[j] = \bigoplus_{k=1}^{t} m^k_{x[k]}[(j-1) \cdot l; j \cdot l]$$

[1]For some string m remember that the notation $m[(j-1) \cdot l; j \cdot l]$ will refer to taking the j'th substring of length l.

The protocol is secure against a corrupt Sender due to never receiving anything, and a corrupt Receiver cannot learn anything from the masks he obtains, since they are random l-bit strings.

Assuming a constant-time OT, and using that the bit-length is constant the running-time of the algorithm is linear in n_1, as the Sender computes n_1 masks, and the Receiver makes n_1 comparisons.

3.1.3 Set intersection

In the basic OT-based private set intersection protocol the Sender will have set $X = \{x_1, \ldots, x_{n_1}\}$ and the Receiver will have a set $Y = \{y_1, \ldots, y_{n_2}\}$, and the two parties want to compute the functionality $f(X, Y) = (\perp, X \cap Y)$.

The simplest way of extending the set inclusion protocol to a private set intersection protocol is having the Receiver shuffle his elements at the beginning of the protocol, and then compute the set inclusion protocol n_2 times, once for each element in Y. This once again increases the bit-length of the masks, to $l = \lambda + \log(n_1) + \log(n_2)$. However this also results in a running-time of $O(n_1 \cdot n_2)$, which is prohibitive for larger sets.

3.2 PSI protocol from [PSZ14]

The asymptotic running-time of the basic OT-based PSI protocol described in Subsection 3.1.3 is too slow for practical use, which made [PSZ14] propose various optimizations, resulting in a fairly efficient protocol. The primary result of [PSZ14] is using two different hashing schemes to reduce the number masks being created and compared.

Intuitively the idea consists of having two parties with one set each, a Sender with set $X = \{x_1, \ldots, x_{n_1}\}$ and a Receiver with set $Y = \{y_1, \ldots, y_{n_2}\}$. As part of the input two hash functions H_1, H_2 are shared, mapping to $\beta = 2 \cdot (1+\epsilon) \cdot n_2$, which is the size required for n_2 elements to be hashed using Cuckoo hashing.

The Sender will hash his set X with H_1, H_2 using simple hashing (see Figure 2.3) to obtain a two-dimensional array $T_1[\,][\,]$ containing two copies of each element in X. To hide the number of elements hashed to a given bucket the Sender will insert a dummy element d_1 until each bucket has size max_β, a value roughly logarithmic in n_1, followed by shuffling the elements in the bucket.

Meanwhile the Receiver will use Cuckoo hashing (see Figure 2.5) to obtain a roughly half full array $T_2[\,]$ with one copy of each of his elements, and insert the dummy element d_2 into all empty positions. As both parties use the same hash functions an element $z \in X \cap Y$ will be hashed to position $pos_z = H_b(z)$ in T_2 by the Receiver, and it will be contained in the pos_z'th bucket in the Sender's array T_1.

By engaging in β set inclusion algorithms, one for each bucket, both the Sender and Receiver will create $\beta \cdot max_\beta$ masks where the Sender will find all masks corresponding to his input elements, insert them in a set V_1 and shuffle V_1. He then sends V_1 to the Receiver, who will find the intersection of the masks and output the corresponding intersection of elements.

3.3 Protocol of [PSZ14]

The exact protocol proposed by [PSZ14] follows the previously outlined idea, however various details were omitted. An important aspect is the use of a stash for Cuckoo hashing, to reduce the chance of failing the hashing to be negligible, which results in a secondary OT being used to mask the elements contained in the stash. The full protocol can be seen in Figure 3.3, and its three steps will be explained in details now.

Hashing the elements: To efficiently force a random ordering on the input sets X and Y the elements will be hashed using two shared hash functions $H_1, H_2 : \{0,1\}^\sigma \to \{1, \ldots, \beta\}$.

The Sender will use simple hashing, Figure 2.3, to obtain a two-dimensional array $T_1[\,][\,]$ containing all his elements. To hide the ordering of his elements he will for each of the β buckets $T_1[\,]$ insert dummy element d_1 until the size of $T_1[\,]$ is $\max_\beta$, followed by randomly permuting the elements contained in the buckets. max_β is a shared value which ensures that the simple hashing algorithm fails with at most negligible probability, determined by the function $F(n_1, \beta)$ estimated in [PSZ14], and if $n_1 = n_2$ is roughly logarithmic in n_1.

The Receiver will use the Cuckoo hashing algorithm from Figure 2.6 to obtain a roughly half full array $T_2[\,]$ and a small stash S containing some elements. To hide how his elements are hashed P_2 inserts the dummy element d_2 into every empty position in $T_2[\,]$ and fills up S until it has size s, followed by randomly permuting the elements in S.

Masking the elements via OT: Equivalently to the basic OT-based PSI protocol the elements will be masked by using random OT, however due to the hashing less masks are generated. The masks will be saved in two two-dimensional arrays $M_{1,h}, M_{2,h}$ which have the same size as T_1 ($\beta \cdot max_\beta$), and two two-dimensional arrays $M_{1,s}, M_{2,s}$ of size $s \cdot n_1$. The $1, 2$ is used to describe whose array it is, that is the Sender uses $M_{1,h}$ and $M_{1,s}$ while the Receiver uses $M_{2,h}$ and $M_{2,s}$. The h and s is used to describe whether the array is for masks corresponding to the hashed or stashed elements, which means $M_{2,h}$ contains masks for elements hashed to T_2 while $M_{2,s}$ contains masks for elements in S.

To mask the elements in T_1 the Sender loops through $1 \leq i \leq \beta$, where he engages in a Rand-$\binom{N}{1}$-OT$^t_{max_\beta \cdot l}$ with the Receiver. From this he obtains the $t \cdot N$ strings

$$(m_1^{i,k}, \ldots, m_N^{i,k}), \text{ where } 1 \leq k \leq t$$

Looping through $1 \leq j \leq max_\beta$ he will then let $x = T_1[i][j]$, which is split into t μ-bit pieces, denoted $x = x[1] \| \cdots \| x[t]$. Using x and the strings obtained from the OT he can generate a mask like in the basic OT-PSI

$$M_{1,h}[i][j] = \bigoplus_{k=1}^{t} m^{i,k}_{x[k]}[(j-1) \cdot l; j \cdot l]$$

The Receiver's computations are roughly equivalent, namely looping through $1 \leq i \leq \beta$ he will let $y = T_2[i]$ which he divides into t μ-bit characters

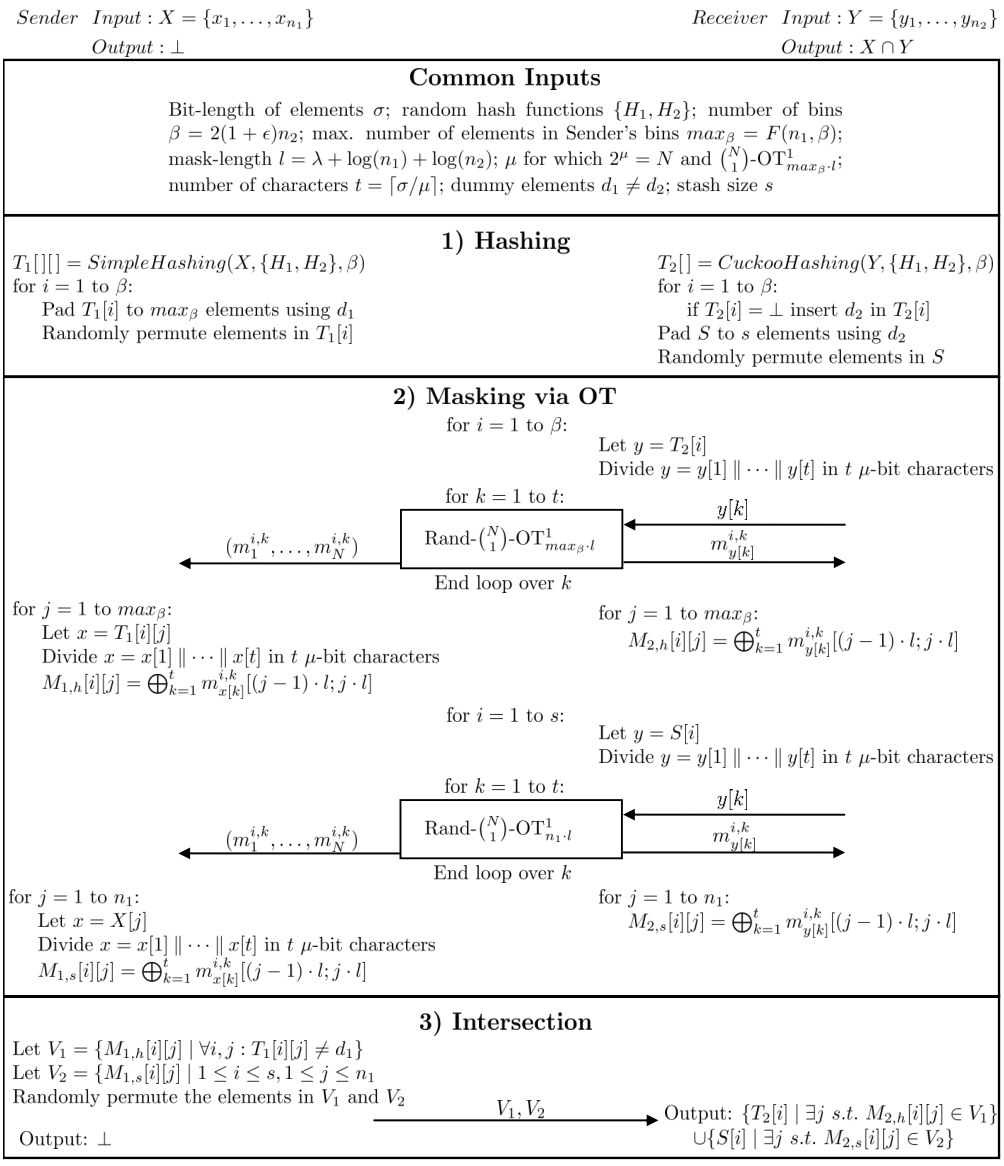

Figure 3.3: Protocol of [PSZ14], copied mostly verbatim from the full paper of [PSSZ15a]

$y = y[1] \,\|\, \cdots \,\|\, y[t]$. The Receiver will then, while looping through $1 \leq k \leq t$, input $y[k]$ to the OT and obtain the string $m^{i,k}_{y[k]}$. After obtaining t strings the Receiver can use them to compute masks equivalently to the Sender, namely looping through $1 \leq j \leq max_\beta$ to compute

$$M_{2,h}[i][j] = \bigoplus_{k=1}^{t} m^{i,k}_{y[k]}[(j-1)\cdot l; j \cdot l]$$

In the same manner both parties will compute masks for the elements in the stash, by looping through $1 \leq i \leq s$ and $1 \leq k \leq t$ to do a Rand-$\binom{N}{1}$-$\text{OT}^1_{n_1 \cdot l}$, where the Receiver obtains his input by using $S[i]$. Then by looping through $1 \leq j \leq n_1$ the Sender will compute the masks, using $X[j] = x = x[1] \,\|\, \cdots \,\|\, x[t]$, as

$$M_{1,s}[i][j] = \bigoplus_{k=1}^{t} m^{i,k}_{x[k]}[(j-1)\cdot l; j \cdot l]$$

The Receiver will use $S[i] = y = y[1] \,\|\, \cdots \,\|\, y[t]$ to compute the masks

$$M_{2,s}[i][j] = \bigoplus_{k=1}^{t} m^{i,k}_{y[k]}[(j-1)\cdot l; j \cdot l]$$

Thus through a Rand-$\binom{N}{1}$-$\text{OT}^{\beta \cdot t}_{max_\beta \cdot l}$, a Rand-$\binom{N}{1}$-$\text{OT}^{s \cdot t}_{n_1 \cdot l}$ and some XOR operations the two parties will have computed masks for their inputs. In particular the Sender has computed $2 \cdot n_1 + s \cdot n_1$ masks, and the Receiver has computed $n_2 \cdot max_\beta + s \cdot n_1$ masks.[2]

Finding the intersection: The last step requires the Sender to send all relevant masks to the Receiver. The masks are defined as two sets, namely

$$V_1 = \{M_{1,h}[i][j] \mid \forall i, j : T_1[i][j] \neq d_1\},\ V_2 = \{M_{1,s}[i][j] \mid 1 \leq i \leq s, 1 \leq j \leq n_1\}$$

V_1 corresponds exactly to the masks of the elements in X, and has size $2 \cdot n_1$. V_2 corresponds to, for each entry in the stash, the mask of each element in X, and has size $s \cdot n_1$. The Sender randomly permutes the elements and sends both sets to the Receiver. After obtaining V_1, V_2 he will compute

$$\{T_2[i] \mid \exists j \ s.t.\ M_{2,h}[i][j] \in V_1\} \cup \{S[i] \mid \exists j \ s.t.\ M_{2,s}[i][j] \in V_2\}$$

which he outputs as the intersection.

3.4 Proof for [PSZ14]

In the paper [PSZ14] no proof of security was included, and thus a proof has been created as part of this thesis.

Proof: Definition 2.8 states two requirements for a protocol to be secure against a static semi-honest adversary. It has to be correct, which means the

[2]Note that neither party is required to compute the masks for the dummy elements, as they will not be compared.

output of a functionality computing $f(X, Y) = (\perp, X \cap Y)$ has to be computationally indistinguishable from the protocol. Secondly two PPT simulators that, given the input and output of a single party, have to produce views that are computationally indistinguishable from the real protocol. The simulator for a corrupt Sender can be seen in Figure 3.4, and for a corrupt Receiver in Figure 3.5.

Correctness Assuming existence of some element $z \in X \cap Y$:

There are two cases, namely that the Receiver hashes z to his Cuckoo array T_2 or to the stash S.

z is hashed to T_2: Without loss of generality assume that z is hashed to position v by the Receiver ($T_2[v] = z$), and the Sender's v'th bin will include a copy of z at position w ($T_1[v][w] = z$).

At some point both parties will compute the mask of the v'th bucket and w'th element in bucket v, meaning that both parties will use z to determine the mask. The Sender will compute

$$M_{1,h}[v][w] = \bigoplus_{k=1}^{t} m_{z[k]}^{v,k}[(w-1) \cdot l; w \cdot l]$$

while the Receiver will input substrings of z to the OT, and obtain the mask

$$M_{2,h}[v][w] = \bigoplus_{k=1}^{t} m_{z[k]}^{v,k}[(w-1) \cdot l; w \cdot l]$$

As these are equivalent both parties will generate the same mask, denoted m_z.

During the third step the Sender will include the mask m_z in the set V_1, since $T_1[v][w] \neq d_1$, and send it to the Receiver. When he compares m_z with the elements in his mask array $M_{2,h}$ he will find that $m_z = M_{2,h}[v][w]$ and include the element $T_2[v] = z$ in his output.

Otherwise if z is hashed to the stash by the Receiver equivalent steps will be computed, and a mask m_z for z will be included in V_2 by the Sender. An equivalent mask will be computed by the Receiver, and included in $M_{2,s}$. This again results in the Receiver outputting z as part of the intersection.

Hence correctness is satisfied.

Using a sequence of hybrid games a number of simulators will be shown, starting with the simulators shown in the Figures 3.4 and 3.5. By making small changes it will be proven that the simulators are computationally indistinguishable from the real protocol.

Simulating the view of corrupt Sender

Game H_0: The simulated execution.

Game H_1: The simulator S_1 no longer uses the functionality f_{OT}, and is instead replaced with the protocol π_{OT}. Given that a simulator exists for π_{OT} the output is guaranteed to be computationally indistinguishable, and thus H_0 and H_1 are indistinguishable.

Game H_2: In this game the trusted party and honest Receiver are removed, and the simulator S_2 is given the Receiver's input Y. S_2 computes the Cuckoo

PSZ14.$\text{SIM}_1(1^\kappa, X, \perp)$

Masking: The simulator $\mathcal{S}$ starts by computing the double array like the real protocol, namely $T_1[\,][\,] = SimpleHashing(X, \{H_1, H_2\}, \beta)$, adds dummy elements and randomly permutes each bin.

OT-phase: For $1 \leq i \leq \beta$ and $1 \leq k \leq t$ $\mathcal{S}$ uses the OT's simulator to obtain N random messages of length $max_\beta \cdot l$ as OT output, denoted $(m_1^{i,k}, \ldots, m_N^{i,k})$, and includes them in the view.

For each bin and for $1 \leq j \leq max_\beta$ let each element in the i'th bin $T_1[i][j] = x = x[1] \parallel \cdots \parallel x[t]$ be used to compute the mask

$$M_{1,h}[i][j] = \bigoplus_{k=1}^{t} m_{x[k]}^{i,k}[(j-1) \cdot l; j \cdot l]$$

In the same manner for $1 \leq i \leq s$ and $1 \leq k \leq t$ $\mathcal{S}$ uses the OT's simulator to obtain N random messages of length $n_1 \cdot l$ as OT output, denoted $(m_1^{i,k}, \ldots, m_N^{i,k})$.

For each bin and for $1 \leq j \leq s$ and for each element $x \in X$, where $x = x[1] \parallel \cdots \parallel x[t]$, be used to compute a mask

$$M_{1,s}[i][j] = \bigoplus_{k=1}^{t} m_{x[k]}^{i,k}[(j-1) \cdot l; j \cdot l]$$

Intersection: The two sets $V_1 = \{M_{1,h}[i][j] \mid \forall i, j : T_1[i][j] \neq d_1\}$ and $V_2 = \{M_{1,s}[i][j] \mid 1 \leq i \leq s, 1 \leq j \leq n_1\}$ are computed, and randomly permuted. Both are included in view as message to the Receiver.

Figure 3.4: Simulator for Sender in [PSZ14]

PSZ14.$\text{SIM}_2(1^\kappa, Y, X \cap Y)$

Masking: The simulator $\mathcal{S}$ starts by computing the array like the real protocol, namely $T_2[\,] = CuckooHashing(Y, \{H_1, H_2\}, \beta)$, and inserts dummy elements into empty spots. The stash is padded with dummy elements until it has size s, and randomly permuted.

OT-phase: For $1 \leq i \leq \beta$ element $T_2[i] = y = y[1] \parallel \cdots \parallel y[t]$ is found. For $1 \leq k \leq t$ $y[k]$ is used as OT input, and 1 random string of length $max_\beta \cdot l$ is obtained from the OT's simulator as OT output, denoted $m^{i,k}_{y[k]}$.

For each bin and for $1 \leq j \leq max_\beta$ the masks

$$M_{2,h}[i][j] = \bigoplus_{k=1}^{t} m^{i,k}_{y[k]}[(j-1) \cdot l; j \cdot l]$$

are computed, and if $y \in X \cap Y$ one random mask from $M_{2,h}[i]$ is saved in V_1.

Equivalently for $1 \leq i \leq s$ element $S[i] = y = y[1] \parallel \cdots \parallel y[t]$ is found. For $1 \leq k \leq t$ y_k is used as OT input, and 1 random string of length $n_1 \cdot l$ is obtained from the OT's simulator as OT output, denoted $m^{i,k}_{y[k]}$.

For each bin and for $1 \leq j \leq n_1$ the masks

$$M_{2,s}[i][j] = \bigoplus_{k=1}^{t} m^{i,k}_{y[k]}[(j-1) \cdot l; j \cdot l]$$

are computed, and if $y \in X \cap Y$ the mask $M_{2,s}[i][r]$ for one $r \in_{\text{R}} \{1, \ldots, n_1\} \setminus R$ is saved in V_2, and r is saved in R (The set X is not shuffled at any point, so to ensure the i'th element is not used to generate masks for two elements this extra step is required).

Intersection: Uniformly random l-bit values are added to V_1 until $|V_1| = 2 \cdot n_1$, and V_1 is randomly permuted. Equivalently uniformly random l-bit values are added to V_2 until $|V_2| = s \cdot n_1$ and V_2 is randomly permuted. The sets V_1, V_2 are included in view as message from the Sender, and $\mathcal{S}$ outputs $X \cap Y$.

Figure 3.5: Simulator for Receiver in [PSZ14]

hash of Y to obtain T_2, S including dummy elements, which it uses as OT input. The Receiver's input to the OT cannot be distinguished by the Sender, and as such cannot be used to distinguish H_1 and H_2.

Note that game H_2 is identical to the real protocol.

Simulating the view of a corrupt Receiver

Game H_0: The simulated execution.

Game H_1: The simulator S_1 no longer uses the functionality f_{OT}, and is instead replaced with the protocol π_{OT}. Given that a simulator exists for π_{OT} the output is guaranteed to be computationally indistinguishable, and thus H_0 and H_1 are indistinguishable.

Game H_2: The trusted party and honest Sender are removed, while the simulator S_2 is given the Sender's input X. From this S_2 computes T_1 as the real protocol, which it uses to generate masks during the OT-phase. These masks are used to compute the sets V_1 and V_2, and are included in view as messages to the Receiver. As the Rand-OT used does not allow the Sender to input anything the only change in transmitted messages are the sets V_1 and V_2. The only meaningful masks for the Receiver are the ones contained in the intersection, and the remaining masks in the sets are uniformly random strings. As he cannot distinguish the two distributions H_1 and H_2 are indistinguishable.

Game H_3: Instead of outputting the intersection obtained as input the Receiver computes the output like the real protocol. As the chance of collisions of elements not contained in the intersection is negligible the two games H_2 and H_3 are indistinguishable.

Note that H_3 is identical to the real protocol.

□

Chapter 4

Optimizations by [PSSZ15a]

[PSSZ15a] proposes three optimizations to the semi-honestly secure PSI protocol of [PSZ14], resulting in significantly better running-times, both asymptotically and in practice.

First the optimizations will be explained, where an easily fixed correctness issue with the third optimization will be shown, followed by showing the full protocol of [PSSZ15a] and then proving it is secure against a static semi-honest adversary. At last the protocol will be analyzed with a malicious adversary which results in three vulnerabilities, where two will be prevented and the new protocol will be proven secure against a malicious Receiver.

The full version of the paper can be seen in [PSSZ15b], which links to the version studied during this thesis.

4.1 Optimizations

The paper suggests three optimizations that reduces the number of masks being generated, increases the number of hash functions and changes the hash functions to a Feistel-like structure, which reduces the bit-length of the hashed elements.

4.1.1 Reducing number of generated masks:

The protocol of [PSZ14] will, for each of the β bins, create $max_\beta = O(\log(n_1))$ masks per bin, resulting in the Receiver creating $\beta \cdot max_\beta + s \cdot n_1$ masks in total. To avoid that [PSSZ15a] proposes a scheme that on average creates a constant number of masks per bin, resulting in $2 \cdot n_1 + s \cdot n_1$ masks by the Sender and $n_2 + s$ masks for the Receiver. The basic idea behind the optimization is reducing the OT-output length to l, and obtaining an intermediate value by XORing all relevant strings together for a given element. To obtain the necessary randomness this intermediate value is hashed using a third hash function, which will give the final mask.

The exact change is as follows. The Sender will use the hashing scheme of Figure 2.4, which no longer limits the number of elements that can be hashed to a single entry. When creating masks for the non-stashed elements the two

parties will engage in a Rand-$\binom{N}{1}$-$\text{OT}_l^{\beta \cdot t}$, instead of a Rand-$\binom{N}{1}$-$\text{OT}_{max_\beta \cdot l}^{\beta \cdot t}$, where the Sender obtains $t \cdot N$ random strings of length l, while the Receiver obtains t random strings of length l.

To compute the masks let the $t \cdot N$ strings the Sender obtains during the i'th Rand-$\binom{N}{1}$-OT_l^t be denoted $(m_1^{i,k}, \ldots, m_N^{i,k})$, where $1 \leq k \leq t$. Let u^i elements be hashed to the i'th bucket, denoted $x_1, \ldots, x_{u^i}$, where the j'th element can be split as $T_1[i][j] = x = x[1] \parallel \cdots \parallel x[t]$. Then an intermediate value

$$S_1^{i,j} = \bigoplus_{k=1}^{t} m_{x[k]}^{i,k}$$

is computed, which is then hashed with hash function H to obtain the mask

$$M_{1,h}[i][j] = H(S_1^{i,j})$$

Equivalently the Receiver has one element hashed to the bucket $T_2[i] = y$, which can be split as $y = y[1] \parallel \cdots \parallel y[t]$ and will be used to obtain the t OT-outputs $(m_{y[1]}^{i,1}, \ldots, m_{y[t]}^{i,t})$. These will be XORed to an intermediate value

$$S_2^i = \bigoplus_{k=1}^{t} m_{y[k]}^{i,k}$$

which will be hashed with H to obtain the final mask for bin i

$$M_{2,h}[i] = H(S_2^i)$$

Note that the Receiver computes exactly one mask for bin i, where he previously computed max_β.

An equivalent computation will be done for the elements in the stash, namely instead of using a Rand-$\binom{N}{1}$-$\text{OT}_{n_1 \cdot l}^{s \cdot t}$ they will use a Rand-$\binom{N}{1}$-$\text{OT}_l^{s \cdot t}$, compute intermediate masks and hash these masks. This will reduce the number of masks the Receiver has to compute to s, while the Sender is still required to compute $s \cdot n_1$ masks for the stash elements.

Correctness Compared to [PSZ14] each mask is now required to be hashed, as they are now dependant on each other. To see this consider the somewhat artificial case where the length of the elements is 2, that is $\sigma = 2$, and where each element will be split into two one-bit pieces, meaning there will be two outputs from each of the two OTs per bin. For a single bin let the four random strings obtained from the OT be denoted r_0^0, r_1^0, r_0^1 and r_1^1, where r_0^i, r_1^i will be the two outputs in the i'th execution of the OT. These strings can be used to generate four valid masks, namely $m_{b,b'} = r_b^0 \oplus r_{b'}^1$.

The primary issue with these dependant masks can be seen in the following equation: $m_{0,0} \oplus m_{0,1} \oplus m_{1,0} \oplus m_{1,1} = 0$. This makes it possible for the Receiver to check whether four given unhashed strings are contained in the same bucket, since the chance that four random masks XOR to 0 is negligible in the statistical security parameter. This also means that if the Receiver obtains three masks he knows are from one bin he can efficiently compute the last - which should

not be possible. Naturally this issue persists when μ and N grow, however the number of strings required increases as well. In particular it is possible to verify whether N^μ elements are contained in the same bucket, and from $N^\mu - 1$ elements to find the last element.

Hashing the values ensures that the generated masks are no longer dependant on each other, and instead uniformly random l-bit strings leaking no information.

Efficiency gain This optimization results in reducing the work done by both the Sender and the Receiver from $\beta \cdot max_\beta$ to β, which is roughly a logarithmic improvement. When benchmarking each optimization [PSSZ15b] reports a factor 10 improvement from reducing the number of masks computed, when testing with $n_1 = n_2 = 2^{20}$.

4.1.2 h-ary Cuckoo Hashing:

The second optimization is varying the number of shared hash functions, namely using the Cuckoo hashing algorithm from Figure 2.7 which uses h hash functions.

As noted during the explanation of Figure 2.7 the 2-ary Cuckoo hashing algorithm requires about 50% of the array to be empty, while 3-ary Cuckoo hashing requires about 9% to be empty, as shown by [FPSS03]. The running-time is dependant on β, which is reduced as the number of hash functions increases. Increasing the number of hash functions also increases the computational overhead of doing Cuckoo hashing for the Receiver and increases the number of masks the Sender is required to both compute and send to the Receiver, thus resulting in a trade-off.

Experimental results by [PSSZ15a] showed that using 3-ary cuckoo hashing was optimal when $n_1 = n_2$, and when $n_1 \gg n_2$ 2-ary cuckoo hashing was optimal. The run-time reduction, according to [PSSZ15a], is roughly a factor $1.2 - 1.3$ when testing sets of equal size.

For the remainder the function $F'(n_2, h)$ will be used to determine the size of β, rather than explicitly writing the size.

4.1.3 Reducing bit-length:

For the last optimization the description given by [PSSZ15a] will be shown, followed by identifying a correctness issue, and a simple fix proposed by the authors of [PSSZ15a] during an e-mail correspondence.

This optimization replaces Cuckoo hashing from Figure 2.7 to that of Figure 2.8, which uses parts of the results from [ANS10] to use Feistel-like functions instead of the hash functions. This makes it possible, for non-stashed element, to reduce the bit-length from σ to $\sigma' = \sigma - \log \beta$, which results in reducing the number of OTs performed per bin from $t = \lceil \frac{\sigma}{\mu} \rceil$ to $t' = \lceil \frac{\sigma'}{\mu} \rceil = \lceil \frac{\sigma - \log \beta}{\mu} \rceil$.

It is important to note that this optimization does not apply to elements in the stash, as the Feistel-like structure requires that the placement of an element is part of its identifier, which is not applicable in the stash. Instead

mask generation for elements in the stash will continue using bit-length σ and splitting each element into t μ-bit pieces.

The simple hashing algorithm used by the Sender will also use the Feistel-like functions, which is simply done by replacing the hash functions.

[PSSZ15a] reported roughly a factor $1.1 - 2$ improvement from replacing the hash functions, when keeping the number of elements constant at 2^{20}, while varying σ. As they also kept the numer of OT outputs constant at $N = 2^8$ the reduction was always $\lfloor \log(n)/8 \rfloor = 2$ less OTs per bin, clearly resulting in less performance increase as σ grows. When $\sigma = 32$ the number of OTs was reduced from $t = 4$ to $t' = 2$, while having $\sigma = 128$ only reduced $t = 10$ to $t' = 8$.

Correctness issue The method used by [ANS10] to reduce the bit-length of hashed elements uses exactly one random function, which means the correctness argument sketched when defining Figure 2.8 holds. However as part of this thesis it was discovered that using two or more random functions breaks the correctness, which is necessary for Cuckoo hashing.

For simplicity assume two random functions are used, f_1, f_2. Then f_1 can be used to hash $x = x_L \parallel x_R$ to $pos = x_L \oplus f_1(x_R)$, where x_R is stored at position pos. From this it is possible to find another left-part of an element by computing $pos \oplus f_2(x_R) = x'_L$ such that $x'_L \parallel x_R$ is hashed correctly while also storing x_R at position pos. Hence when using x_R as input to the OT the parties obtain a valid mask for both $x_L \| x_R$ and $x'_L \| x_R$, which breaks the correctness of the protocol. Extending this to h random functions clearly results in obtaining one valid mask that checks whether one of h elements are hashed to position pos.

In an email correspondence with the authors of [PSSZ15a] this issue was brought up, where the authors acknowledged the issue and said they were made aware of it after their paper was accepted. At the same time they suggested fixing it by using an additional $\log(h)$ bits for each element to identify which hash function is being used. This in particular means the stored element is $x_R \parallel k$ at position pos when computing $pos = x_L \oplus f_k(x_R)$. As $\log(h)$ is small this is only a minor decrease in efficiency.

When using the Feistel-like structure to hash elements this trick of concatenating the hash function identifier at the end will be assumed to be a part of the hashing, and hence will not be mentioned explicitly. Note that this means Figure 2.8 is not an exact match of what happens, and instead a simplification. Also note that when referencing the Feistel-like structures they will not be explicitly written, and are referenced as $\{H_1, \ldots, H_h\}$.

4.2 Protocol of [PSSZ15a]

The full protocol of [PSSZ15a] can be seen in Figure 4.1. It uses the same idea as [PSZ14] with the optimizations sketched above, and as such no further explanations will be included.

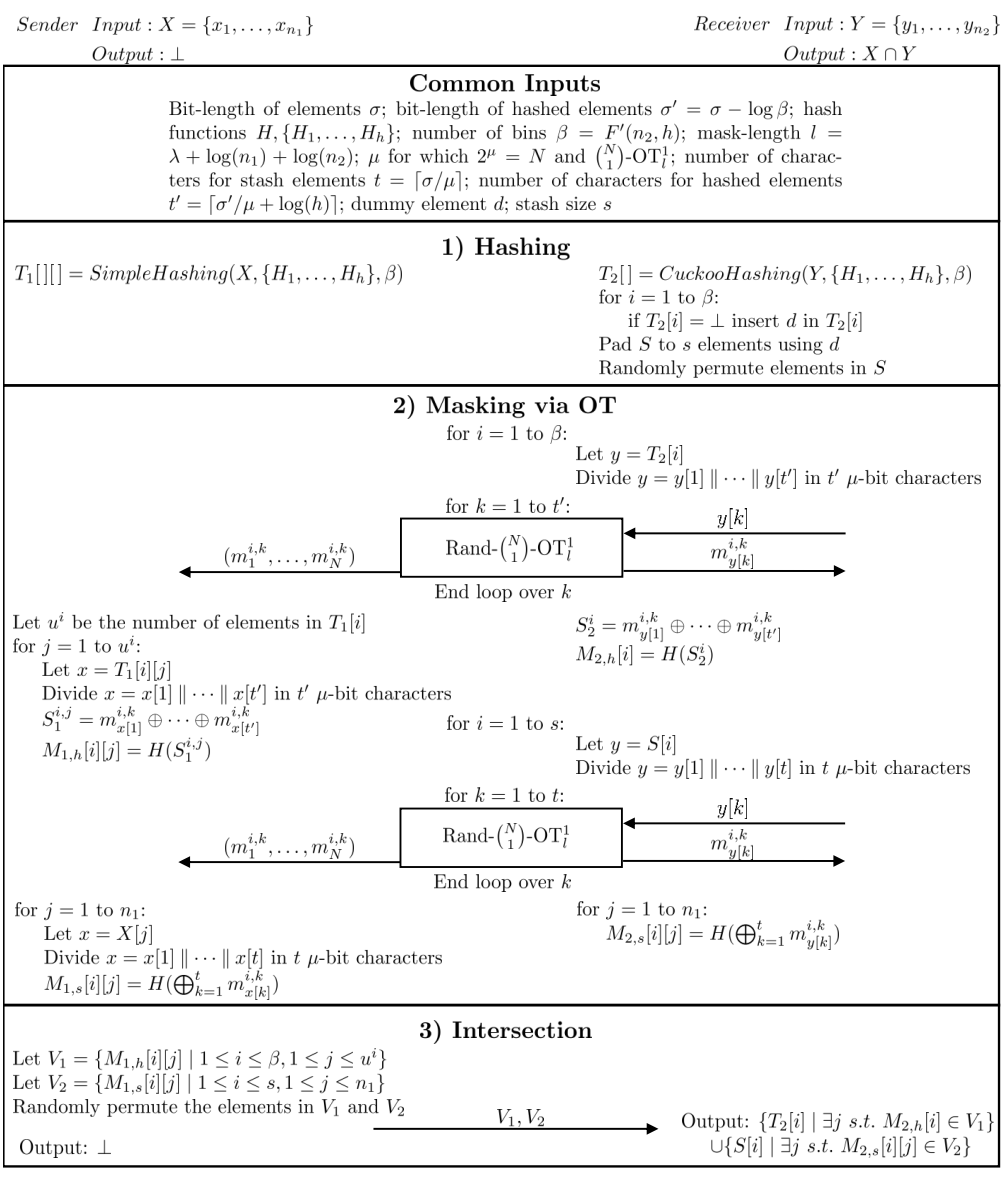

Figure 4.1: Protocol of [PSSZ15a]

4.3 Proof for [PSSZ15a]

A proof of security was not included in [PSSZ15a], but has been created as part of this thesis.

Proof:

Definition 2.8 states two requirements for a protocol to be secure against a static semi-honest adversary. It has to be correct, which means the output of the functionality $f_\cap$ computing $f_\cap(X, Y) = (\perp, X \cap Y)$ has to be computationally indistinguishable from the protocol. Secondly two PPT simulators need to be constructed such that they produce a view that is computationally indistinguishable from the real protocol, given the input and output of a single party. The simulator for the Sender can be seen in Figure 4.2, and for the Receiver in Figure 4.3

Correctness Assuming existence of some element $z \in X \cap Y$:

Correctness is split into two cases, either the Receiver hashes z into the cuckoo hash table, or it is placed in the stash. The two cases will be treated separately.

Assume that $z = z_L \parallel z_R$ is hashed to the Cuckoo table where it will be at position $v = H_g(z) = z_L \oplus f_g(x_R)$ for some random function $f_g \in F$, meaning that $T_2[v] = z_R$ for the Receiver[1]. The Sender will have z_R as the w'th element in bin v, resulting in $T_1[v][w] = z_R$.

During mask creation the content of bin v will at some point be used to determine the masks. The Receiver will use the substrings of z_R to determine his input to the OT, resulting in t' masks $m^{v,k}_{z_R[k]}$ for $1 \leq k \leq t'$. To create the mask of z he will XOR all the obtained masks together, resulting in

$$S_2^v = \bigoplus_{k=1}^{t'} m^{v,k}_{z_R[k]}$$

This element will be hashed to obtain the final mask

$$M_{2,h}[v] = H(S_2^v)$$

Meanwhile the Sender will obtain the $t' \cdot N$ masks denoted $(m_1^{v,k}, \ldots, m_N^{v,k})$ for $1 \leq k \leq t'$. When creating masks he will eventually use the w'th element, which is z_R. This will result in obtaining the value

$$S_1^{v,w} = \bigoplus_{k=1}^{t'} m^{v,k}_{z_R[k]}$$

which will be hashed to obtain the mask

$$M_{1,h}[v][w] = H(S_1^{v,w})$$

[1]Remember that the identifier for g is required which means what is actually stored is $T_2[v] = z_R \parallel g$, however to ease notation this has been omitted. This applies for the Sender as well.

When the Sender sends his masks he will include $M_{1,h}[v][w]$ in V_1 and send it to the Receiver. When he compares $M_{2,h}[v]$ to the elements in V_1 he will be guaranteed to find $M_{2,h}[v] \in V_1$, resulting in the Receiver outputting z as part of the intersection.

If z is inserted into the stash let the Receiver have it at position v, such that $S[v] = z$.

When creating masks for the stashed elements at some point both parties will use the v'th stash element. This result in the Receiver inputting substrings of z to the OT, and thus obtaining the t masks $m_{z[k]}^{v,k}$ for $1 \leq k \leq t$. These will be XORed and then hashed to obtain the mask

$$M_{2,s}[v] = H(\bigoplus_{k=1}^{t} m_{z[k]}^{v,k})$$

At the same time the Sender will create n_1 masks per stash element. In particular he will for the v'th stash element obtain the $t \cdot N$ masks $(m_1^{v,k}, \ldots, m_N^{v,k})$ for $1 \leq k \leq t$. As z is contained in his set he will compute the mask

$$M_{1,s}[v][j] = H(\bigoplus_{k=1}^{t} m_{z[k]}^{v,k})$$

for some j less than n_1.

When sending masks for stashed elements the Sender will send $M_{1,s}[v][j]$, which the Receiver will find equivalent to $M_{2,s}[v]$, which means he will output z as part of the intersection.

View of PSSZ15.Sim$_1$ The view of PSSZ15.Sim$_1$ is computationally indistinguishable from the real protocol:

To prove indistinguishability of PSSZ15.Sim$_1$ it needs to be proven that the OT output is distributed uniformly random, and used together with X to generate the sets V_1, V_2, that as a result consists of $h \cdot n_1$ and $s \cdot n_1$ uniformly random l-bit strings.

The simulated OT output is generated by the simulator guaranteed to exist for the OT, which means it is computationally indistinguishable from the real protocol, as otherwise the OT would not be secure. This can be seen when generating $\beta \cdot N \cdot t'$ random l-bit values, for the hashed elements, and $s \cdot N \cdot t$ random l-bit values for the elements in the stash.

The masks are computed equivalently to the real protocol, and since the computation uses the masks generated by the OT's simulator they are computationally indistinguishable from the real protocol. Since this is true for both the masks in V_1 and V_2 the simulator for a corrupt Sender is computationally indistinguishable from the real protocol.

View of PSSZ15.Sim$_2$ The view of PSSZ15.Sim$_2$ is computationally indistinguishable from the real protocol:

To prove indistinguishability of PSSZ15.Sim$_2$ three kinds of messages need to be distributed correctly, namely the OT input, the OT output and the sets V_1, V_2.

PSSZ15.$\text{SIM}_1(1^\kappa, X, \perp)$

Masking: Use simple hashing to obtain the two-dimensional array like the real protocol, $T_1[\,][\,] = SimpleHashing(X, \{H_1, \ldots, H_h\}, \beta)$.

OT-phase: For $1 \leq i \leq \beta$ and $1 \leq k \leq t'$ use the OT's simulator to obtain N random messages of length l as OT output, denoted $(m_1^{i,k}, \ldots, m_N^{i,k})$, and include them in the view.

For each bin and for $1 \leq j \leq u^i$ set $T_1[i][j] = x = x[1] \parallel \cdots \parallel x[t']$, and use it to generate the mask $M_{1,h}[i][j] = H(\bigoplus_{k=1}^{t'} m_{x[k]}^{i,k})$.

Equivalently for $1 \leq i \leq s$ and $1 \leq k \leq t$ use the OT's simulator to obtain N random messages of length l as OT output, denoted $(m_1^{i,k}, \ldots, m_N^{i,k})$, and include them in the view.

For each possible stash element and for $1 \leq j \leq n_1$ find the j'th element $X[j] = x = x[1] \parallel \cdots \parallel x[t]$, and compute $M_{1,s}[i][j] = H(\bigoplus_{k=1}^{t} m_{x[k]}^{i,k})$.

Intersection: Compute the sets $V_1 = \{M_{1,h}[i][j] \mid 1 \leq i \leq \beta, 1 \leq j \leq u^i\}$ and $V_2 = \{M_{1,s}[i][j] \mid 1 \leq i \leq s, 1 \leq j \leq n_1\}$, randomly permute them and include them in view as message to the Receiver.

Figure 4.2: Simulator for Sender in [PSSZ15a]

The OT input needs to consist of the elements in Y and the dummy element d, mixed together. This is done by using the same hashing as the real protocol, namely by using Cuckoo hashing followed by inserting d into all empty spots, and filling up the stash S until it contains s elements, and shuffle it. This results in the simulator doing exactly as the real protocol, and hence cannot be distinguished from it.

In the real protocol the OT output is $\beta \cdot t' + s \cdot t$ uniformly random strings of length l that are computationally indistinguishable from the OT output in PSSZ15.SIM_2, due to the OT's simulator.

The two sets V_1, V_2 need to be of the correct size, namely $h \cdot n_1$ for V_1 and $s \cdot n_1$ for V_2. They also need to contain the masks corresponding to the intersection, with the remainder of the elements being uniformly random elements. The elements contained in the intersection are included as they are computed, and the remaining elements are uniformly random l-bit values that cannot be distinguished from the real protocol, meaning that PSSZ15.SIM_2 is computationally indistinguishable form [PSSZ15a].

At last it is obvious that both simulators run in PPT, which concludes the proof. □

PSSZ15.$\textsc{Sim}_2(1^\kappa, Y, X \cap Y)$

Masking: Use the cuckoo hashing algorithm to compute the one-dimensional array $T_2[\,] = CuckooHashing(Y, \{H_1, \ldots, H_h\}, \beta)$, and insert d into all empty positions. Fill the stash S until it has size s with d, and randomly permute S.

OT-phase: For $1 \leq i \leq \beta$ let $T_2[i] = y = y[1] \,\|\, \cdots \,\|\, y[t']$, and for $1 \leq k \leq t'$ include $y[k]$ as OT input, and use the OT's simulator to obtain 1 random message of length l as OT output, denoted $m^{i,k}_{y[k]}$.

Use y to compute the mask $M_{2,h}[i] = H(\bigoplus_{k=1}^{t'} m^{i,k}_{y[k]})$, and if $y \in X \cap Y$ save $M_{2,h}[i]$ in set V_1.

Equivalently For $1 \leq i \leq s$ let $S[i] = y = y[1] \,\|\, \cdots \,\|\, y[t]$, and for $1 \leq k \leq t$ include $y[k]$ as OT input, and use the OT's simulator to obtain 1 random message of length l as OT output, denoted $m^{i,k}_{y[k]}$.

Use y to compute the mask $M_{2,s}[i] = H(\bigoplus_{k=1}^{t} m^{i,k}_{y[k]})$, and if $y \in X \cap Y$ save $M_{2,s}[i]$ in set V_2.

Intersection: Insert random l-bit values in V_1 until $|V_1| = h \cdot n_1$ and randomly permute V_1. Equivalently insert random l-bit values in V_2 until $|V_2| = s \cdot n_1$ and randomly permute V_2. Include V_1, V_2 in view as message from the Sender, and output $\{T_2[i] \mid \exists j \ s.t. M_{2,h}[i][j] \in V\}$ combined with the set $\{S[i] \mid \exists j \ s.t. \ M_{2,s}[i][j] \in V_2\}$.

Figure 4.3: Simulator for Receiver in [PSSZ15a]

4.4 Malicious adversary against [PSSZ15a]

During work prior to this thesis three vulnerabilities were discovered when the PSI protocol of [PSSZ15a] was analyzed with a malicious adversary, rather than the semi-honest adversary it is created to be secure against. Methods to prevent two of the three attacks have been found during this thesis. The vulnerabilities are as follows:

First, the random OT being used in the protocol is secure against a semi-honest adversary, and as such any malicious attack depending on the OT will likely extend to a vulnerability in the PSI protocol. As the OT is being used as a functionality no specific attack will be mentioned, and it can trivially be prevented by using a maliciously secure random OT functionality instead, and as such will not be discussed further.

Secondly a malicious Receiver is able to verify whether some small subset of elements are contained in the Sender's set, despite not being included in his input. This attack can be prevented by using secret sharing and symmetric-key encryption, as will be explained shortly.

The last is a malicious Sender that can influence correctness of the protocol, by making the output dependant on how the Receiver hashes his elements. A second attack influencing correctness that is a result of the countermeasure to the malicious Receiver has also been found. No working countermeasures have been found for this attack.

4.4.1 Malicious Receiver

This section will consist of four parts, namely an exact algorithm to attack the PSI protocol as a malicious Receiver combined with arguments to show that the attack is possible and efficient. Afterwards it will be shown how to prevent this by using Shamir's secret sharing scheme to enforce correct behaviour from the Receiver. Following this optimizations to increase the efficiency of the scheme will be shown, as the secret sharing imposes a quadratic overhead, together with minor tweaks to ensure correctness and security. At last the new protocol is proven secure against a malicious Receiver.

Description of attack: An exact algorithm to attack the PSI protocol as a malicious Receiver can be seen in Figure 4.4, which will be used to argue about the severity of the attack.

Intuitively the attack uses, assuming two hash functions, that the Cuckoo array $T_2[\,]$ consists of roughly 50% empty spots after inserting n_2 elements that constitute as the Receiver's input. The basic idea behind the attack is taking some element e and hashing it using both H_1 and H_2. If $T_2[H_i(e)] = \perp$ then e can be inserted into T_2 without removing elements from Y. When creating masks through the random OT a valid mask will be created for e rather than the dummy element that ought to be at e's position. When the Sender sends his mask set V_1 the Receiver can check whether $e \in V_1$, and if true he knows e is contained in the Sender's input set X except with negligible probability. If $e \notin V_1$ he knows $e \notin X$ as well. The same idea applies to the stash S, where

```
1: procedure RECEIVER.ATTACK(Y)
2:     Hash elements in Y as usual, resulting in T_2[ ] and S[ ]
3:     for interesting elements e do
4:         for Each hash function H_i do
5:             if T_2[H_i(e)] = ⊥ then
6:                 Insert e into T_2[H_i(e)]
7:             else
8:                 if |S| < s then
9:                     Insert e into S
10:    Complete protocol as usual
```

Figure 4.4: An attack on the PSI protocol of [PSSZ15a] as a malicious Receiver

elements can be inserted until it has size s, to create masks and check whether said elements are included in X. This results in providing the Receiver with full knowledge of whether e is contained in the Sender's input.

Due to the size restriction on both T_2 and S a malicious Receiver is bounded in the number of elements he can obtain knowledge of. Using $h = 2$ the array T_2 is roughly half full, while the size of S is a small constant, which results in the Receiver learning inclusion of about n_2 elements he ought to not learn about. This number is decreased as h increases.

The existence of an element e that can be inserted into T_2 when using $h = 2$ (the argument extends to $h > 2$ as the percentages are based on how Cuckoo hashing works) is as follows: for an element $e \notin Y$ the Receiver will hash it twice, using H_1, H_2. Both times he will check if $T_2[H_i(e)] = \perp$, and if true insert e, which results in him learning whether e is contained in X. The Cuckoo hash array is at most half full and the hash functions are modelled as random oracles, resulting in an upper bound of $\frac{1}{4}$'th chance of $T_2[H_i(e)] \neq \perp$ when hashing with both hash functions. Repeating this a polynomial $p(\kappa)$ number of times, using different elements, results in $(\frac{1}{4})^{p(\kappa)}$ chance of not finding an element hashing to an empty spot. As this probability is negligible that means a malicious Receiver is able to efficiently insert some element into T_2, resulting in learning whether said element is contained in X.

Countermeasure: As part of this thesis Shamir's secret sharing scheme paired with symmetric-key encryption will be proposed to prevent this attack. The basic idea to preventing a malicious Receiver obtaining more information is making the dummy elements necessary.

The Sender generates a key k and uses a $(\beta+s-n_2, \beta+s)$-threshold Shamir's secret sharing scheme to split k into $\beta + s$ shares. He then computes the $\beta + s$ dummy element masks and uses these as a one-time pad encryption on the key-shares, resulting in the values $(r_1, \ldots, r_{\beta+s})$. The Sender then computes the two sets V_1', V_2' obtained as usual and encrypts them using k resulting in $V_1 = E(k, V_1')$ and $V_2 = E(k, V_2')$. Then he sends $V_1, V_2, (r_1, \ldots, r_{\beta+s})$ to the Receiver.

This means the Receiver is required to obtain $\beta + s - n_2$ dummy element

masks to decrypt the one-time pad encryption of $\beta + s - n_2$ key-shares to reconstruct the key. From this he can decrypt the sets and compute the intersection.

A formal description of this can be seen in Figure 4.5.

Ensuring correctness, security and efficiency: The modified protocol has a few subtle changes to ensure that correctness is preserved and the asymptotic overhead is not too high. The first change is the length of l, which has to be increased to ensure that an l-bit value can be used as one-time pad encryption of the key-shares. This is done by increasing the length from $l = \lambda + \log(n_1) + \log(n_2)$ to $l = \max(\kappa, \lambda + \log(n_1) + \log(n_2))$. With less length than the new l it might be possible to brute-force the encryption, clearly violating the security of the protocol.

The second issue is a result of the quadratic running-time of Shamir's secret sharing scheme. The number of dummy elements, or empty positions in T_2, determines the input size to Shamir's secret sharing scheme, and thus the running-time is significantly increased as the input size, and hence number of dummy elements, grows.

To prevent this issue the number of hash functions will once again change, but instead of being predetermined as a function of the difference in set sizes it will be computed as a function dependant on the Receiver's input size n_2. This function will keep the number of dummy elements constant, at the cost of sending slightly more masks in the set V_1, as its size is $h \cdot n_1$.

Proof of modified protocol: At last a proof to show the modified protocol is secure against a malicious Receiver. Note that the proposed protocol is not secure against a malicious Sender, and thus can only be proven for a corrupt Receiver. The protocol is secure against a semi-honest Sender, but as that proof is closely related to the proof shown for [PSSZ15a], except adding a few more computations, it will be omitted.

Proof:

To prove the modified protocol secure against a malicious Receiver Definition 2.9 states a simulator $\mathcal{S}$ needs to be constructed. $\mathcal{S}$ needs to extract the input of the corrupt party $\mathcal{A}$ to use it as input to the trusted party computing the functionality $f_\cap$, and ensure the output corresponds to what the trusted party returns. This needs to be done while simulating a view that is indistinguishable from the real protocol.

The functionality $f_\cap$ computing the intersection needs a maximum input size, defined as n_1 for the Sender and n_2 for the Receiver, defined prior to inputting the sets. If the input exceeds those limits it will output $abort_i$, where party i has input too many elements.

The simulator for a corrupt Receiver can be seen in Figure 4.6. The basic idea is playing the role of a trusted party for the OT, and using the Receiver's OT inputs to compute his cuckoo array and stash, as he cannot change his input after the OT. From this the simulator $\mathcal{S}$ inputs the remaining set to the functionality. If $f_\cap$ outputs an intersection the corresponding masks will be encrypted, excluding wrongly hash elements, and otherwise zeroes of the same

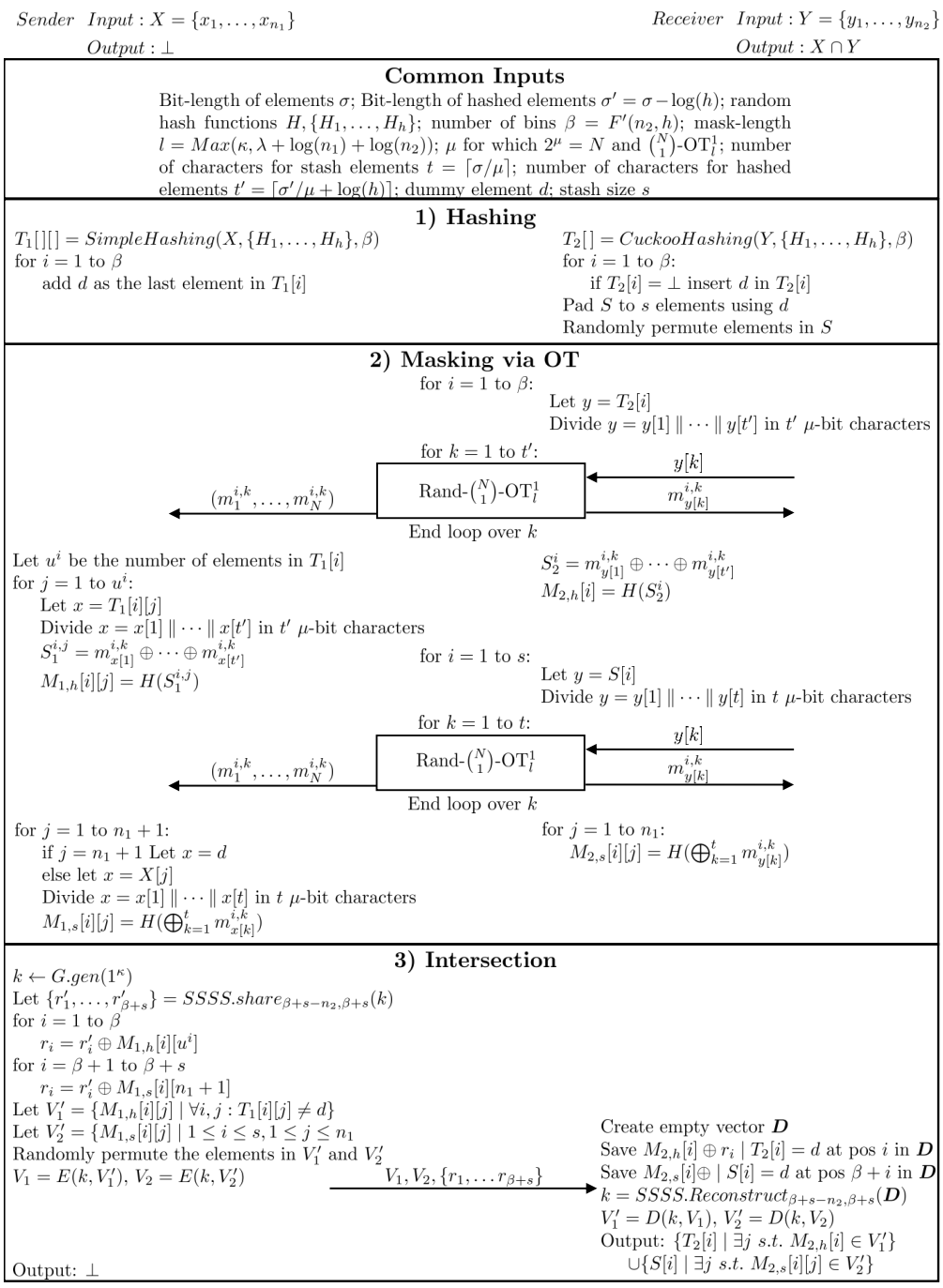

Figure 4.5: Protocol of [PSSZ15a], modified to be secure against a malicious Receiver

length will be encrypted.

To prove security the transmitted messages in the simulator need to be computationally indistinguishable from the real protocol, which means the OT output, the encrypted sets and the one-time pad encrypted elements. The output also needs to be consistent with the functionality's output.

$\mathcal{S}$'s OT output is computationally indistinguishable from the real protocol, as it uses the simulator guaranteed to exist for the OT to generate the messages.

This leaves the two encrypted sets V_1, V_2 together with the random values.

The indistinguishability of the real and ideal world is dependant on the encryption scheme satisfying IND-CPA, meaning without the key $\mathcal{A}$ has only a negligible probability of decrypting the sets, which means it also relies on the secret sharing scheme hiding the key. This splits into two cases depending on whether the adversary $\mathcal{A}$ has input at least $\beta + s - n_2$ dummy elements to the OT or not.

If he did input at least $\beta + s - n_2$ dummy elements he can compute at least $\beta + s - n_2$ masks of dummy elements that can be used to decrypt the one-time encryption of the key-shares, and obtain at least $\beta + s - n_2$ key-shares. As he used $\beta + s - n_2$ dummy elements his input set consists of at most n_2 elements, that have been input to the functionality which resulted in the simulator obtaining the intersection. This means V_1, V_2 consists of masks (and random values), and that he can obtain the encryption key from the one-time pad encrypted elements, and thus $\mathcal{A}$ cannot distinguish whether he is in the ideal or real world when properly computing the intersection.

If he did not input $\beta + s - n_2$ dummy elements he cannot learn the key from the values $r_1, \dots r_{n+s}$, as they are uniformly random values. However this is exactly as the real protocol, as the key-shares would be uniformly random values, and he needs $\beta + s - n_2$ elements to reconstruct the key. Since he does not have that he knows nothing about the key, and cannot decrypt the two sets. Thus the encryption of zeroes from the simulator and encrypted masks in the real world cannot be distinguished, as he cannot decrypt either.

To formally prove that a cheating adversary's views are computationally indistinguishable in the real and ideal world a reduction to the encryption scheme's property IND-CPA will be made. By assuming a distinguisher D that can distinguish the simulated and hybrid views a dinstinguisher D_E will be made that breaks IND-CPA:

Upon receiving auxiliary input z D_E emulates the view of a Sender until the last step where it has to send two encrypted sets and the key-shares. D_E then creates four sets. The first two are created as the protocol, namely a set containing masks corresponding to D_E's simple hashing array, and one set containing masks corresponding to the stash, both randomly permuted. The two last sets contain only zeroes. D_E then inputs the four sets to the encryption oracle such that the two sets containing masks is the first input, and the two sets containing zeroes is the second input. The encryption oracle returns two encrypted sets V_1'' and V_2'', which D_E gives to D together with $\beta + s$ uniformly random values. D outputs a guess b on whether the sets contain masks or

LO16.$\text{SIM}_2(z)$

The simulator invokes $\mathcal{A}$ on auxiliary input z

OT-phase: The adversary $\mathcal{A}$ outputs $\beta \cdot t'$ strings as OT-input. For each string $w[k]$ the simulator uses it to reconstruct the adversary's array T_2. It also uses the simulator guaranteed to exist for the Random OT to obtain N strings. These are saved in array $W_h[i][k]$, where $1 \leq i \leq \beta$ and $1 \leq k \leq t'$. The simulator returns $W_h[i][k][w[k]]$ to $\mathcal{A}$.

Equivalently $\mathcal{A}$ outputs $s \cdot t$ strings corresponding to his stash, from which the simulator reconstructs the stash S. For each string $w[k]$ the simulator obtains N random strings from the OT's simulator saved in $W_s[i][k]$, for $1 \leq i \leq s$ and $1 \leq k \leq t$, where it returns $W_s[i][k][w[k]]$ as OT-output to $\mathcal{A}$.

Constructing sets: From the constructed T_2, S all dummy elements are removed, and the resulting set is input to the functionality $f_\cap$.

If the output is the intersection Z the masks for each $z \in Z$ are constructed by finding its position, except for any wrongly hashed element (meaning that element $z \in T_2$ at position i will have $H_k(z) \neq i$ for all hash functions. Any element that has been inserted into such a position cannot be contained in the intersection, as the Sender is guaranteed to not create the mask). If $T_2[i] = z$ the mask will be constructed from strings in $W_h[i]$, as the real protocol and it will be saved in V_1'. Otherwise if $S[i] = z$ the strings in $W_s[i]$ will be used, and the mask will be saved in V_2'. V_1' is filled with random masks until it has size $h \cdot n_1$ and V_2' is filled with random values until it has size $s \cdot n_1$. At the end both sets are randomly permuted.

Otherwise if $f_\cap$ outputs $abort_2$ V_1' is filled with $h \cdot n_1$ zeroes of length l, and V_2' is filled with $s \cdot n_1$ zeroes of length l.

Sending response: A key k is generated and used to encrypt V_1' and V_2' to obtain V_1, V_2. If V_1' and V_2' contain masks k is split into $\beta + s$ shares using Shamir's secret sharing scheme, and one-time pad encryption is used to encrypt each key-share with the mask of a dummy element, like the real protocol, to obtain the elements $r_1, \ldots, r_{\beta+s}$. Otherwise if V_1', V_2' are encryptions of zero $r_1, \ldots, r_{\beta+s}$ are obtained as uniformly random l-bit values.

$V_1, V_2, \{r_1, \ldots, r_{\beta+s}\}$ is sent to the adversary $\mathcal{A}$.

Figure 4.6: A simulator against a malicious corrupt Receiver, as modified version of [PSSZ15a], proposed as part of this thesis

zeroes, and D_E outputs b as well.

If D, and hence D_E, guesses correct with a non-negligible probability it means one of two things. Either D has broken IND-CPA of the encryption scheme, or can distinguish the secret sharing of the key encrypted under one-time pad with the dummy elements from uniformly random elements. As the latter is unconditionally secure it reduces to breaking IND-CPA with non-negligible probability.

□

4.4.2 Malicious Sender

This section will include a detailed description of the attack a malicious Sender is able to perform on the protocol described by [PSSZ15a], and a secondary attack that is a result of preventing a malicious Receiver. Whether either of the issues can be efficiently prevented is an open problem.

A malicious Sender is able to influence the correctness of the protocol by replacing the masks in V_1 and V_2 to not include all $h+s$ masks of each element. As the size of V_1 and V_2 is predetermined, and can efficiently be verified by the Receiver, just adding or removing elements is not sufficient. The attack basically consists of having two elements e, e' where the Sender computes all $h+s$ masks of each element, denoted $m_1^e, \ldots, m_{h+s}^e$ and $m_1^{e'}, \ldots, m_{h+s}^{e'}$. If a malicious Sender includes m_1^e and $m_2^{e'}, \ldots, m_{h+s}^{e'}$ in the sets V_1, V_2 it corresponds to having $\frac{1}{h+s}$ chance of e being in X and $\frac{h+s-1}{h+s}$ chance of e' being in X, which cannot be modelled in the ideal world. In particular the Receiver's output is now dependant on how he hashes his elements, as each mask is created at a position specified by the hash functions.

The added step of sending one-time padded key-shares allows a malicious Sender to encrypt wrong shares, resulting in the Receiver reconstructing the wrong key, and hence obtaining uniformly random values, rather than masks. These elements would be a function of how the Receiver hashed his elements, as the key-shares he can obtain are dependant on where the dummy elements are inserted in his data structure. This does not constitute for input substitution, as the probability of having $h+s$ valid masks for the possible element is negligible, which means it does not satisfy correctness of the protocol.

Chapter 5

PSI using garbled Bloom filters by [DCW13]

[DCW13] proposes two PSI protocols based on their variant of Bloom filters, named garbled Bloom filters (see Figure 2.10), and oblivious transfer. Their first protocol is secure against a semi-honest adversary, while the second is an enhanced version of the first protocol meant to be secure against a malicious adversary. At first the basic protocol will be sketched, followed by focusing on the enhanced version and its flaws.

5.1 Semi-honestly secure protocol from [DCW13]

The full protocol is shown in Figure 5.1. It uses four pre-shared values, namely n as their set sizes, $h = \kappa$ to determine the number of hash functions, H as the set of h hash functions and m as the size of a Bloom filter supporting h hash functions and n elements. It starts by having the Sender create an (m, n, h, H, κ)-garbled Bloom filter (see Figure 2.10), GBF_1, representing his input set X, while the Receiver creates a (m, n, h, H)-Bloom filter (see Figure 2.9), BF_2, representing his input set Y.

The only interaction between the two parties is computing m OTs. In the i'th OT the Sender will input $(r_i, GBF_1[i])$, where r_i is a uniformly random κ-bit value, and the Receiver will use $BF_2[i]$ as his selection bit. From the m random κ-bit strings obtained from the m OTs the Receiver will build a garbled Bloom filter $GBF_{X \cap Y}$. For each element $y \in Y$ he will check if y is represented in $GBF_{X \cap Y}$, and if true y will be output as part of the intersection.

Proof of correctness and proof sketch of security is shown in [DCW13], and will thus be omitted from this thesis.

5.2 Enhanced protocol from [DCW13]

An enhanced version of the semi-honestly secure protocol was also included in [DCW13], and was meant to be secure against a malicious adversary. To obtain that the authors of [DCW13] identified two vulnerabilities in the basic protocol

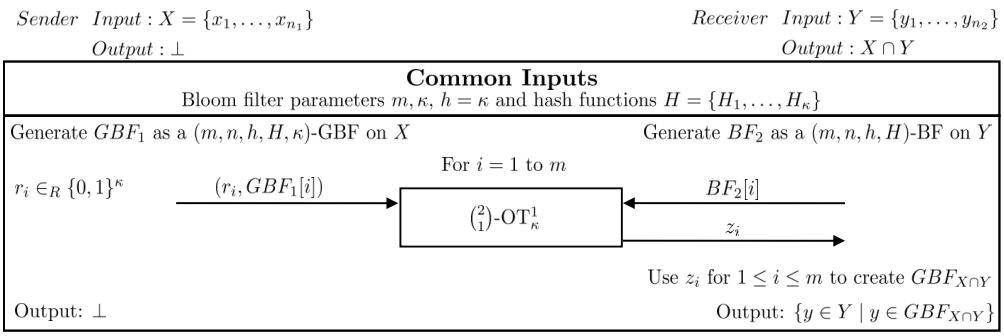

Figure 5.1: Semi-honestly secure PSI protocol from [DCW13]

that a malicious adversary can exploit, followed by restricting his behaviour. However, as will be seen shortly, two vulnerabilities still persist.

5.2.1 Attacks on semi-honestly secure protocol from [DCW13]

Two attacks were sketched in [DCW13]:

The first attack consists of the Receiver inputting 1 into all m OTs, rather than using his Bloom filter entries as selection bits. This provides him with the Sender's full garbled Bloom filter, and as a result the Receiver is able to efficiently compute whether a given element is contained in the Sender's input set X.

This vulnerability is prevented by having the Sender use a symmetric-key block cipher that encrypts each entry in his garbled Bloom filter prior to inputting them into the OT. The key is split using a secret sharing scheme, where the Receiver is able to reconstruct the key by inputting enough zeroes to the OT.

Specifically the Sender generates a uniformly random key k using a symmetric-key encryption generator, which is split using a $(m/2, m)$-threshold secret sharing scheme to obtain the shares $(k_1, \ldots, k_m)$, where $m = 2\kappa n$. For the i'th OT the Sender will input the pair

$$(k_i, E(k, GBF_1[i]))$$

which causes the Receiver to either obtain a key share or an encrypted entry of the Garbled Bloom filter. Informally the argument for security is as follows: If the Receiver inputs at least $m/2$ zeroes he will be able to reconstruct k, which makes it possible to decrypt the encrypted garbled Bloom filter entries and hence construct the new garbled Bloom filter $GBF_{X \cap Y}$, from which he is able to find the intersection. If, however, he inputs less than $m/2$ zeroes he won't be able to reconstruct k, and the encrypted values will provide no information due to the encryption scheme satisfying IND-CPA.

The second identified vulnerability consists of a malicious Sender inputting wrong key-shares shares to the OT, since the Receiver cannot distinguish whether the Sender sends correct or wrong key-shares. If he sends wrong key-shares the Receiver will obtain wrong garbled Bloom filter entries when decrypting, which

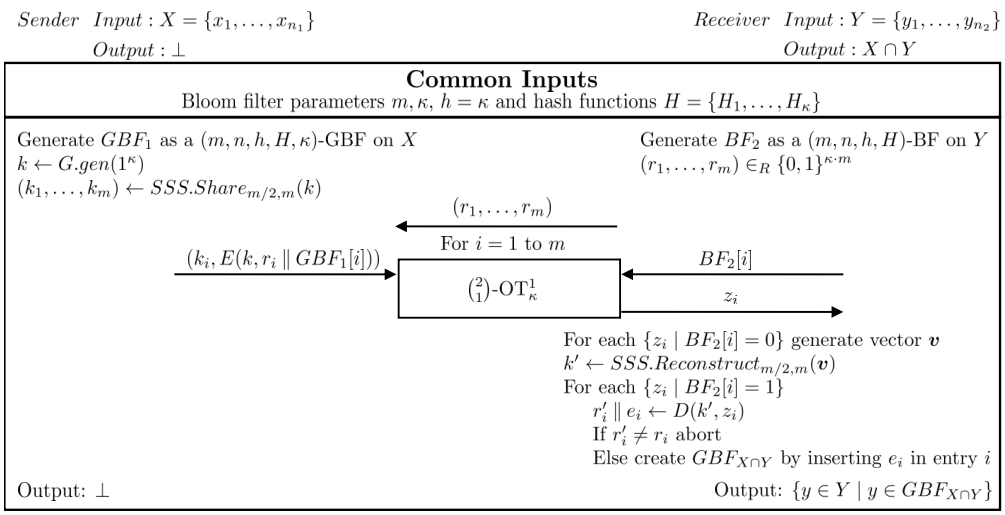

Figure 5.2: Enhanced PSI protocol from [DCW13]

causes him to construct a wrong $GBF_{X \cap Y}$, and as a result correctness is no longer satisfied.

To prevent this attack [DCW13] makes the Receiver send m κ-bit uniformly random strings $(r_1, \ldots, r_m)$ to the Sender. The sender will then use the pair

$$(k_i, E(k, r_i \| GBF_1[i]))$$

as input to the OT instead.

[DCW13] argues for security by saying that the probability the Receiver obtains a different key $k' \neq k$ and that the i'th entry decrypts to a string where the first κ bits are equal to r_i is negligible in κ, which means he should be able to detect whether he obtains the correct key or not.

For completeness the enhanced protocol of [DCW13] can be seen in Figure 5.2.

5.2.2 Attacks on enhanced protocol from [DCW13].

Despite including a proof sketch for security of their enhanced protocol issues still exist. In fact two attacks will be shown as results of thesis, where the first makes it possible to determine whether a given element is contained in the Receiver's input, while the second makes the Sender's input dependant on the Receiver's input, where both attacks are performed as a malicious Sender. Whether either of the issues can be efficiently prevented is an open problem.

Check whether one element is contained in the Receiver's input

The first attack is based on selective failure, which reveals the Receiver's OT input. Intuitively the attack is dependant on an OT protocol providing the Receiver with information of exactly one of the Sender's inputs. In particular for some invocation of the OT the Sender inputs the pair (OT_0, OT_1), and the Receiver will input the bit b, from which the Receiver obtains OT_b and

```
1: procedure SENDER.CHECKCONTAINMENT(e)
2:     Create GBF_1, k, (k_1, ..., k_m) as usual
3:     Obtain random strings (r_1, ..., r_m) from the Receiver
4:     for i = 1 to m do
5:         if H_j(e) = i for some H_j ∈ H then
6:             Input (0, E(k, r_i ‖ GBF_1[i])) to the OT
7:         else Input (k_i, E(k, r_i ‖ GBF_1[i])) to the OT
8:     return ⊥
```

Figure 5.3: Attack on enhanced protocol from [DCW13] as a malicious Sender

can verify correctness of OT_b. The Receiver does not obtain any knowledge of OT_{1-b}, and as such cannot verify whether OT_{1-b} is constructed correctly. This selective failure attack is performed by having the Sender input some wrong pairs of (r, OT_1), where r is meaningless data and OT_1 is the correctly formed input. If the Receiver obtains r he will abort outputting $\perp$, and otherwise he will correctly output the intersection.

The selective failure can be used to determine whether a given element e is contained in the Receiver's input, and make him either output an intersection containing e or abort outputting $\perp$. If $e \in Y$ then the honest Receiver is guaranteed to satisfy

$$BF_2[H_j(e)] = 1 \ \forall H_j \in H$$

If $e \notin Y$ with high probability there will be a hash function $H_j \in H$ where $BF_2[H_j(e)] = 0$. Both of these statements follow directly from analysis of Bloom filters, see [DCW13].

If the Sender wants to check whether $e \in Y$, that is if element e is contained in the Receiver's input set Y, he will do as in Figure 5.3. As sketched the Sender will use the knowledge that $BF_2[H_j(e)] = 1$ and input a meaningless string as OT output if the Receiver inputs 0. This will ensure that the Receiver abandons the protocol with high probability if $e \notin Y$, and if $e \in Y$ he is guaranteed to finish the protocol and output an intersection that includes e. The chance that he successfully finishes the protocol with $e \notin Y$ is negligible in κ.

It is important to understand the leakage of a protocol suffering from selective failure, as it cannot be securely composed. As shown either the enhanced PSI protocol of [DCW13] leaks whether an element is contained in the input (and output), or it aborts outputting $\perp$. This clearly breaks the protocol's composability, as using the output as input in a subsequent protocol either leaks to the adversary that e is contained in the input, or it is meaningless. Since composability of protocols is a security goal in itself clearly the protocol cannot be deemed secure under a standard definition when it is vulnerable to selective failure.

Make Sender's input dependant on the Receiver's input

A malicious Sender can make his input dependant on the Receiver's input, by having the Receiver decrypt one of two different garbled Bloom filters. This is

possible due to [DCW13] not providing any requirements for their encryption scheme, which means using one-time pad encryption should be secure. This is, however, not true as the Sender can make the Receiver accept two different keys k_0, k_1 that share a significant number of points after being secret shared. The Receiver will obtain either k_0 or k_1 as a function of two Bloom filter entries.

For simplicity a small example will be used to describe the attack (where $m = 4$), followed by extending it to larger m.

When $m = 4$ it will be assumed that the Sender knows two entries in the Receiver's Bloom filter (without loss of generality assume that he knows $BF_2[1] = 0$ and $BF_2[2] = 1$), and he wants to make the Receiver's output dependant on his last two entries. He will do this by generating two different garbled Bloom filters GBF_0 and GBF_1, after obtaining $(r_1, \dots r_4)$ from the Receiver. Then he generates two different keys k_0 and k_1 under the constraints

$$E(k_0, r_i \parallel GBF_0[i]) = E(k_1, r_i \parallel GBF_1[i]) \forall i \leq 4$$

Note that this is easy, when one-time pad encryption is used.[1]

He then splits k_0 into 4 shares $(ks_0^1, \dots ks_0^4)$. Using ks_0^1 and k_1 he reconstructs a new secret sharing $(ks_1^1, \dots, ks_1^4)$ where $ks_0^1 = ks_1^1$. Note that ks_0^2 and ks_1^2 are irrelevant, as the Sender knows the Receiver will not obtain a key-share in the second OT.

In the first two OTs the Sender will input the pairs

$$(ks_0^1, E(k_0, r_1 \parallel GBF_0[1])) \ , \ (ks_0^2, E(k_0, r_2 \parallel GBF_0[2]))$$

and for the third and fourth he will input, respectively,

$$(ks_0^3, E(k_1, r_3 \parallel GBF_1[3])) \ , \ (ks_1^4, E(k_0, r_4 \parallel GBF_0[4]))$$

Note the difference in key-shares and encryption being input during the third and fourth OT, and since the Sender knows the Receiver will not input 0 into the second OT he inputs a random key-share.

The Receiver will, from this, obtain

$$ks_0^1, \ E(k_0, r_2 \parallel GBF_0[2]), \ ks_b^{4-b} \text{ and } E(k_b, r_{4-b} \parallel GBF_b[4-b])$$

for some choice of $b \in \{0, 1\}$.[2] This results in reconstructing k_b, which decrypts the two garbled Bloom filter entries to obtain GBF_b, and hence the Sender's input is dependant on the Receiver's input.

This example can be extended for larger m, where knowledge is assumed of $m-2$ entries, where the Sender is able to make the Receiver decrypt to one of two garbled Bloom filters depending on the remaining two entries in his Bloom filter. Without knowledge of $m - 2$ Bloom filter entries the Sender cannot construct two sets of keys-shares that guarantee decrypting the Receiver's random values correctly.

[1] Except for the parts that encrypt the third and fourth entries the two keys are equal. For the third entry, ignoring r_3, $GBF_0[3] \oplus GBF_1[3]$ is the difference in the two keys. Equivalently for the fourth entry.

[2] $4 - b$ is either 3 or 4

Chapter 6

Private Set Operations from [DC16]

[DC16] proposes two protocols that with minor modifications can be adapted to compute four private set operations, namely union (PSU), intersection (PSI), union-cardinality (PSU-CA) and intersection-cardinality (PSI-CA). The first protocol is secure against a semi-honest adversary, and will be the only one included in this thesis, while the second protocol is secure against a malicious adversary.

First a description of the PSU protocol will be given, followed by the modifications required to obtain a PSI, PSU-CA and a PSI-CA protocol, as they are proposed by [DC16]. At last five problems have been found during this thesis, where each issue will be shown and, unless otherwise specified, solutions found as part of this thesis will be proposed.

The semi-honestly secure protocols allow slightly different leakage compared to what has been described so far. The standard allowed leakage for set operation protocols allows both parties to obtain both set sizes, however [DC16] allows that the Receiver is allowed to learn the Sender's set size, while the Sender should learn nothing of the Receiver's set size.

6.1 PSU protocol

Intuitively the set union protocol from [DC16] consists of three steps, where h hash functions $\{H_1, \ldots, H_h\}$ are assumed to be shared. The Receiver is given the Sender's set size, while the Sender knows nothing of the Receiver's set size. The full protocol can be seen in Figure 6.1.

In the first step the Receiver generates an additively homomorphic encryption-key pair (pk, sk). He then computes a Bloom filter of his elements BF_2, where he inverts all entries in BF_2 to obtain the inverted Bloom filter IBF_2. For each entry i in IBF_2 he will encrypt it using pk, to obtain

$$EIBF_2[i] = E(pk, IBF_2[i])$$

resulting in the encrypted inverted Bloom filter $EIBF_2$, where he sends $(pk, EIBF_2)$ to the Sender. This basically means for each entry i in BF_2 where $BF_2[i] = 1$

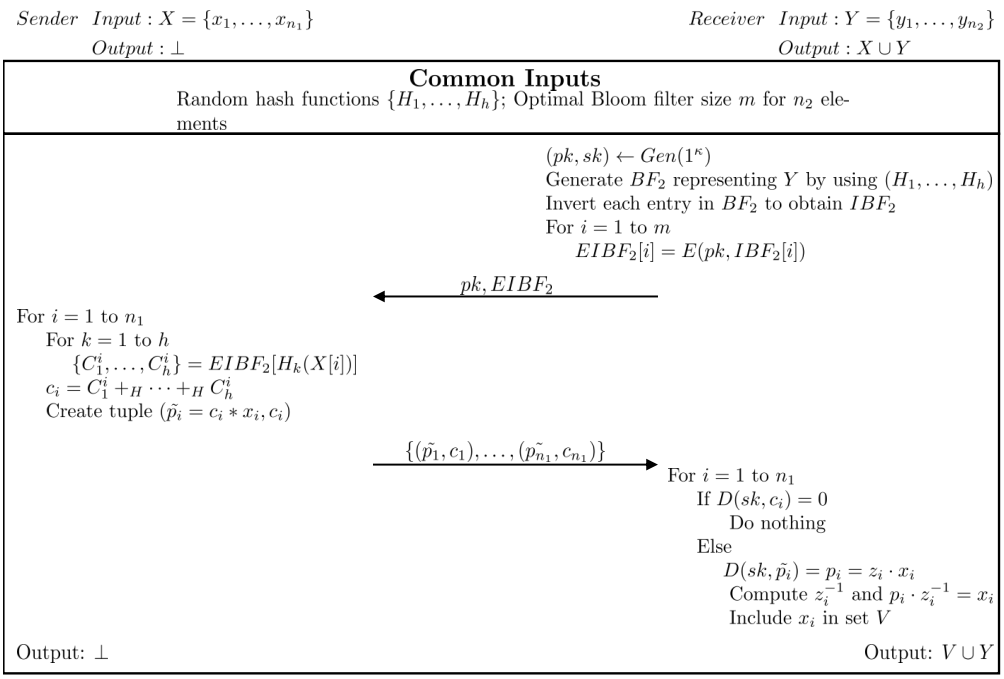

Figure 6.1: PSU protocol from [DC16]

the Receiver sends $E(pk, 0)$, and for each entry where $BF_2[i] = 0$ he sends $E(pk, 1)$. Note that $EIBF_2$ has not been randomized, meaning it is ordered as the corresponding Bloom filter.

In the second step the Sender will for each element $x_i \in X$ use the h hash functions to find the encrypted values $\{C_1^i, \ldots, C_h^i\}$:

$$C_k^i = EIBF_2[H_k(x_i)] \mid 1 \leq k \leq h, x_i \in X$$

For each x_i he computes

$$c_i = (C_1^i +_H \cdots +_H C_h^i)$$

That is using the function defined by the encryption scheme to be homomorphic addition of the ciphertexts. For an element in the union, but not intersection, this results in the Sender including some C_k^i where $EIBF_2[H_k(x_i)] = E(pk, 1)$ as the Receiver does not have anything hashed to position $H_k(x_i)$, and as such the decryption of c_i returns a non-zero element. For an element in the intersection this is not true, and thus $D(sk, c_i) = 0$. At last he will send the tuples

$$(\tilde{p_i} = c_i * x_i, c_i) \ \forall x_i \in X$$

to P_2. Remember that $c_i * x_i$ means the scalar multiplication of the decrypted text with x_i, that is $D(sk, c_i * x_i) = z_i \cdot x_i$ where $z_i = D(sk, c_i)$.

In the third step the Receiver will for each received tuple $(\tilde{p_i}, c_i)$ check whether $D(sk, c_i) = 0$. If true it trivially follows that $D(sk, \tilde{p_i}) = 0$ as well, due to the decrypted value of $\tilde{p_i}$ being a multiplication of $D(sk, c_i)$. Otherwise if $D(sk, c_i) = z_i \neq 0$ the Receiver will compute

$$D(sk, \tilde{p_i}) = D(sk, c_i * x_i) = z_i \cdot x_i$$

Inverting z_i and multiplying z_i^{-1} to $z_i \cdot x_i$ results in x_i, which can be saved in a set V. At the end $V \cup Y$ will be output as the union.

The proof of security can be found in [DC16], and while the basic idea for correctness will be sketched together with the simulator for a corrupt Sender the full proof will be omitted from this thesis, as the second problem is related to the simulator for a corrupt Receiver (which will be sketched later).

The correctness is based on the inverted Bloom filter, which results in the Receiver only being able to obtain the x_i's where $x_i \notin Y$. For an $x_i \notin Y$ the homomorphic addition will include at least one entry where

$$D(sk, EIBF_2[H_k(x_i)]) = 1$$

and as such in the tuple being send in step three

$$D(sk, c_i) = z_i \neq 0$$

and thus

$$D(sk, c_i * x_i) = z_i \cdot x_i \neq 0$$

From this x_i can be computed and included in the output.

The simulator for a corrupt Sender has to create m encrypted values that are indistinguishable from an encrypted inverted Bloom filter that is used in real protocol. This can trivially be done by creating m encryptions of zero and referencing IND-CPA, Definition 2.6, as the corrupt Sender only obtains the public key pk. From this fake $EIBF_2$ the simulator computes n_1 messages like the real protocol and includes them in its view, followed by outputting $\perp$.

6.2 PSI protocol

[DC16] suggests changing the PSU protocol to compute PSI by modifying three parts of the protocol. The first is no longer inverting the Receiver's Bloom filter, which results in sending $E(pk, BF[i])$ for each $1 \leq i \leq m$, as the encrypted Bloom filter EBF_2. Secondly a multiplicative homomorphic encryption scheme will be used, which causes c_i to be computed slightly different, namely

$$c_i = C_1^i \cdot_H \cdots \cdot_H C_h^i \text{ where } C_k^i = EBF_2[H_k(x_i)]$$

At last the value $\tilde{p_i}$ is computed as $\tilde{p_i} = c_i \cdot_H E(pk, x_i)$.

The correctness relies on $D(sk, c_i) = 1$ if each C_k^i is an encryption of 1, which is guaranteed to be true for elements in the Receiver's set. As c_i decrypts to one $c_i \cdot_H E(pk, x_i)$ will decrypt to x_i, and as such the Receiver can find the intersection. If c_i decrypts to 0, which will be the case for all elements not in the intersection, $D(sk, c_i \cdot_H E(pk, x_i)) = 0$ as well, and thus hides the Sender's set. The security arguments rely on the same arguments as the set union protocol, and will not be mentioned here.

6.3 PSU-CA and PSI-CA

[DC16] uses the observation

$$|X \cup Y| = |X| + |Y| - |X \cap Y|$$

to define two almost equivalent protocols computing the union and intersection cardinality. As the union size can trivially be computed given the intersection size (and vice versa) it can be considered as one protocol. Their proposal is using their PSU protocol where the sender excludes the message $\tilde{p_i}$, and thus only sends c_i. This means if $D(sk, c_i) = 0$ the x_i used to compute c_i is contained in the intersection but not union, and if $D(sk, c_i) \neq 0$ the corresponding x_i is contained in the union but not intersection.

The correctness follows trivially from the PSU protocol, and the security will be sketched shortly.

6.4 Problems

Five issues have been identified in [DC16] as part of this thesis, that will be explained shortly. The first issue is evident in their proof of security for the PSU protocol. The second issue is lacking enough randomness. The third issue is their suggestion of multiplicative homomorphic encryption scheme. The fourth issue is minor leakage during their cardinality protocols, and the last issue is related to leaking an upper bound on the Receiver's set size. A solution will be sketched for all five problems.

6.4.1 Security proof

The security proof for PSU from [DC16] is roughly as follows, for a corrupt Receiver:

The simulator for a corrupt Receiver is given $Y, X \cup Y$ and $|X|$, from which it can compute the intersection size $|I|$ and the union size $|U|$. It creates a key pair (pk, sk), and an encrypted inverted Bloom filter $EIBF_2$ of its input Y and includes $(pk, EIBF_2)$ as a part of its view. Secondly it constructs $|I|$ encryptions of 0 and $|U| \setminus |Y| = |X| \setminus |I|$ encryptions of $c_i = C_1^i +_H \cdots +_H C_h^i$ (where C_k^i is computed as usual) and the corresponding $\tilde{p_i}$ for each element in $X \cup Y$. These $|X|$ encryptions are included as part of the view, and the simulator outputs $X \cup Y$.

Their argument of security relies on being able to compute the Sender's $|X| \setminus |I|$ elements in the union, while the $|I|$ elements in the intersection are indistinguishable from the real protocol, due to IND-CPA security of the encryption scheme.

During this thesis one issue is found to be present in this proof. It is using IND-CPA, Definition 2.6, to argue for privacy of the elements contained in the intersection. The definition states that an adversary cannot distinguish between the encryption of two different values, given the public key. However in this case the adversary (Receiver) also knows the secret key, and as such the

property does not apply to the problem. The proof should have used circuit privacy, Definition 2.7, which states that the adversary cannot distinguish between newly encrypted values and values obtaining through the homomorphic $Eval$, even given the secret key. Using circuit privacy it follows that the $|I|$ encryption of zeroes cannot be used to distinguish between the real and ideal world. Additionally [DC16] uses Paillier's additively homomorphic encryption scheme [Pai99] as example, which satisfies this property.

6.4.2 Lacking randomness

The homomorphic $Eval$ provides no new randomness, which means if both a Sender and Receiver compute some $\tilde{p_i}$ the encrypted message $\tilde{p_i}$ will be equivalent for both of them.

When the Sender computes his n_1 ciphertext pairs $\{(\tilde{p_1}, c_1), \ldots, (\tilde{p_{n_1}}, c_{n_1})\}$ during the PSU protocol the intersection is meant to be hidden from the Receiver. This is not the case, since for each $y \in Y$ the Receiver can compute the corresponding $\tilde{p_y}$ and check whether it is contained in the n_1 elements from the Sender. If $\tilde{p_y} \in \{\tilde{p_1}, \ldots, \tilde{p_{n_1}}\}$ he knows that $y \in X$, and hence he learns the intersection.

This can be prevented by creating two new encryptions of zero for each pair of c_i and $\tilde{p_i}$. These encryptions $d_1 = E(pk, 0), d_2 = E(pk, 0)$ can be added using the homomorphic addition, such that the Sender sends $(\tilde{p_i} +_H d_1, c_i +_H d_2)$ for new d_1, d_2 for each pair. This results in ciphertexts that are uniformly random values from which the Receiver cannot determine which of his elements are contained in X.

6.4.3 Multiplicative homomorphic encryption

As part of the PSI protocol it is suggested that ElGamal, [Gam84], can be used as multiplicative homomorphic encryption scheme, which means the operations are computed modulo p. In the PSI protocol [DC16] wants to encrypt the number zero, however ElGamal can only encrypt elements in $\mathbb{Z}_p^*$, and since $0 \notin \mathbb{Z}_p^*$ the PSI protocol cannot use ElGamal.

This issue can be fixed in one way, which was hinted at by [DC16]. Their idea consists of reusing the additively homomorphic encryption scheme for computing the intersection. This is done by having the Receiver compute an encrypted inverted Bloom filters of his input, and sending it together with a public key to the Sender, who computes the c_i's as specified in the PSU protocol, meaning

$$c_i = EIBF_2[H_1(x_i)] +_H \cdots +_H EIBF_2[H_h(x_i)]$$

To compute $\tilde{p_i}$ some randomness is required to hide elements not contained in the intersection. This means $\tilde{p_i} = c_i * r_i +_H E(pk, x_i)$, where r_i is a uniformly random string. The message send to the Receiver then contains $(\tilde{p_i}, c_i)$, as the PSU protocol.

This results in two cases, either $D(sk, c_i) = 0$ which means $D(sk, c_i * r_i) = 0$ and thus

$$D(sk, \tilde{p_i}) = D(sk, c_i * r_i +_H E(pk, x_i)) = D(sk, E(pk, x_i)) = x_i$$

This can only happen if $EIBF_2[H_k(x_i)] = 0$ for all hash functions, which only happens if $x_i \in Y$, except with negligible probability. Otherwise if $D(sk, c_i) \neq 0$ then $D(sk, c_i * r_i)$ is a uniformly random number from which the homomorphic addition of x_i leaks nothing about x_i, and thus the Receiver learns nothing.

6.4.4 Cardinality protocols

For a corrupt Sender the security follows directly from the PSU protocol, however for a corrupt Receiver their argument is not correct. They split it into two cases, namely if $D(sk, c_i) = 0$ nothing is leaked (which is correct), and if $D(sk, c_i) = z_i \neq 0$ they argue that z_i does not leak anything about the x_i used to compute c_i. However this is not correct, as z_i is the number of hash functions evaluated on x_i where $BF_2[H_k(x_i)] = 0$. This means a Receiver can verify whether any other element q is contained in $X \setminus Y$ by evaluating all h hash functions on the inverted Bloom filter and compute $q' = \sum_{k=1}^{h} IBF_2[H_k(q)]$. If q' is different from all z_i's obtained from the Receiver q is guaranteed to not be contained in X, thus clearly leaking something.

This can trivially be fixed by having the Sender multiply each c_i with a uniformly random number r_i to obtain $c_i' = c_i * r_i$, and send c_i' for each i. This results in the decrypted z_i either being zero or a random number, where both cases leak nothing about X.

6.4.5 Size of the Receiver's input

As the first transmitted message the Receiver sends m encrypted values to the Sender, where each value is correlated to his Bloom filters. The number m can be used to efficiently compute an upper bound on the Receiver's set size, which is not allowed leakage. At the start of the protocol the Receiver has to compute m to be optimal, that means as small as possible, for his input size n_2. The formula used to compute the number of bins m is

$$m \geq n_2 \cdot \log(e) \cdot \log(1/\epsilon) \tag{6.1}$$

The paper suggests using $\epsilon \approx 2^{-100}$, which results in

$$\log(e) \cdot \log(1/\epsilon) \approx 145$$

Inserting that into Formula 6.1 gives $m \geq n_2 \cdot 145$ which can be rewritten to

$$n_2 \leq \frac{m}{145}$$

which means that the Receiver's set size at most one 145'th of the number of elements he sends.

This issue can be fixed by using the standard model for PSI, that is to allow both set sizes to be leaked.

Bibliography

[ALSZ15] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. More efficient oblivious transfer extensions with security for malicious adversaries. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I*, volume 9056 of *Lecture Notes in Computer Science*, pages 673–701. Springer, 2015.

[ANS10] Yuriy Arbitman, Moni Naor, and Gil Segev. Backyard cuckoo hashing: Constant worst-case operations with a succinct representation. In *51th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2010, October 23-26, 2010, Las Vegas, Nevada, USA*, pages 787–796. IEEE Computer Society, 2010.

[Blo70] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[BR93] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In Dorothy E. Denning, Raymond Pyle, Ravi Ganesan, Ravi S. Sandhu, and Victoria Ashby, editors, *CCS '93, Proceedings of the 1st ACM Conference on Computer and Communications Security, Fairfax, Virginia, USA, November 3-5, 1993.*, pages 62–73. ACM, 1993.

[CT10] Emiliano De Cristofaro and Gene Tsudik. Practical private set intersection protocols with linear complexity. In Radu Sion, editor, *Financial Cryptography and Data Security, 14th International Conference, FC 2010, Tenerife, Canary Islands, January 25-28, 2010, Revised Selected Papers*, volume 6052 of *Lecture Notes in Computer Science*, pages 143–159. Springer, 2010.

[DC16] Alex Davidson and Carlos Cid. Computing private set operations with linear complexities. *IACR Cryptology ePrint Archive*, 2016:108, 2016. URL of analyzed version: `http://eprint.iacr.org/2016/108/20160210:220813`.

[DCW13] Changyu Dong, Liqun Chen, and Zikai Wen. When private set intersection meets big data: An efficient and scalable protocol. *IACR*

Cryptology ePrint Archive, 2013:515, 2013. URL of analyzed version: http://eprint.iacr.org/2013/515/20130827:085046.

[Dem12] Laurent Demanet. Chaper 3: Interpolation. http://ocw.mit.edu/courses/mathematics/18-330-introduction-to-numerical-analysis-spring-2012/lecture-notes/MIT18_330S12_Chapter3.pdf, 2012. [Online; accessed 07-March-2012].

[FPSS03] Dimitris Fotakis, Rasmus Pagh, Peter Sanders, and Paul G. Spirakis. Space efficient hash tables with worst case constant access time. In Helmut Alt and Michel Habib, editors, *STACS 2003, 20th Annual Symposium on Theoretical Aspects of Computer Science, Berlin, Germany, February 27 - March 1, 2003, Proceedings*, volume 2607 of *Lecture Notes in Computer Science*, pages 271–282. Springer, 2003.

[Gam84] Taher El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In G. R. Blakley and David Chaum, editors, *Advances in Cryptology, Proceedings of CRYPTO '84, Santa Barbara, California, USA, August 19-22, 1984, Proceedings*, volume 196 of *Lecture Notes in Computer Science*, pages 10–18. Springer, 1984.

[Gol04] Oded Goldreich. *The Foundations of Cryptography - Volume 2, Basic Applications*. Cambridge University Press, 2004.

[HEK12] Yan Huang, David Evans, and Jonathan Katz. Private set intersection: Are garbled circuits better than custom protocols? In *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012*. The Internet Society, 2012.

[HL10] Carmit Hazay and Yehuda Lindell. *Efficient Secure Two-Party Protocols - Techniques and Constructions*. Information Security and Cryptography. Springer, 2010.

[IKNP03] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In Dan Boneh, editor, *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, volume 2729 of *Lecture Notes in Computer Science*, pages 145–161. Springer, 2003.

[KK13] Vladimir Kolesnikov and Ranjit Kumaresan. Improved OT extension for transferring short secrets. *IACR Cryptology ePrint Archive*, 2013:491, 2013.

[KMRS14] Seny Kamara, Payman Mohassel, Mariana Raykova, and Seyed Saeed Sadeghian. Scaling private set intersection to billion-

element sets. In Nicolas Christin and Reihaneh Safavi-Naini, editors, *Financial Cryptography and Data Security - 18th International Conference, FC 2014, Christ Church, Barbados, March 3-7, 2014, Revised Selected Papers*, volume 8437 of *Lecture Notes in Computer Science*, pages 195–215. Springer, 2014.

[KMW09] Adam Kirsch, Michael Mitzenmacher, and Udi Wieder. More robust hashing: Cuckoo hashing with a stash. *SIAM J. Comput.*, 39(4):1543–1561, 2009.

[Orl15] Claudio Orlandi. Cryptographic computation foundations lecture notes, 2015. Not publicly available.

[Pai99] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In Jacques Stern, editor, *Advances in Cryptology - EUROCRYPT '99, International Conference on the Theory and Application of Cryptographic Techniques, Prague, Czech Republic, May 2-6, 1999, Proceeding*, volume 1592 of *Lecture Notes in Computer Science*, pages 223–238. Springer, 1999.

[PR01] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. In Friedhelm Meyer auf der Heide, editor, *Algorithms - ESA 2001, 9th Annual European Symposium, Aarhus, Denmark, August 28-31, 2001, Proceedings*, volume 2161 of *Lecture Notes in Computer Science*, pages 121–133. Springer, 2001.

[PSSZ15a] Benny Pinkas, Thomas Schneider, Gil Segev, and Michael Zohner. Phasing: Private set intersection using permutation-based hashing. In Jaeyeon Jung and Thorsten Holz, editors, *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.*, pages 515–530. USENIX Association, 2015.

[PSSZ15b] Benny Pinkas, Thomas Schneider, Gil Segev, and Michael Zohner. Phasing: Private set intersection using permutation-based hashing. *IACR Cryptology ePrint Archive*, 2015:634, 2015. URL of analyzed version: `http://eprint.iacr.org/2015/634/20150702:114451`.

[PSZ14] Benny Pinkas, Thomas Schneider, and Michael Zohner. Faster private set intersection based on OT extension. In Kevin Fu and Jaeyeon Jung, editors, *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.*, pages 797–812. USENIX Association, 2014.

[Sha79] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.

[Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th Annual Symposium on Foundations of Computer Science, Toronto, Canada, 27-29 October 1986*, pages 162–167. IEEE Computer Society, 1986.